

WebSphere Message Broker



Message Flows

Version 6 Release 0

WebSphere Message Broker



Message Flows

Version 6 Release 0

Note

Before using this information and the product it supports, read the information in the Notices appendix.

Third Edition (March 2006)

This edition applies to IBM® WebSphere Message Broker Version 6.0 and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2000, 2006. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this topic collection. v

Part 1. Developing message flows . . . 1

Developing message flows 3

Message flows overview	3
Designing a message flow	59
Managing message flows	122
Defining message flow content	135
Developing ESQL	145
Developing Java	285
Developing message mappings	302
Defining promoted properties	365
Collecting message flow accounting and statistics data	374
Developing user exits	380
Configuring aggregation flows	384
Configuring timeout flows	402
Using an MQGet node in a request-response flow	407

Part 2. Deploying 417

Deploying 419

Deployment overview	419
Deploying a message flow application	429
Deploying a broker configuration.	436
Deploying a publish/subscribe topology	438
Deploying a publish/subscribe topics hierarchy	439
Checking the results of deployment	441
Canceling a deployment that is in progress	443
Renaming objects that are deployed to execution groups	445
Removing a deployed object from an execution group	445

Part 3. Debugging 449

Testing and debugging message flow applications 451

Debugging a message flow	451
------------------------------------	-----

Part 4. Reference 477

Message flows 479

Message flow preferences	479
Description properties for a message flow	479

Built-in nodes	482
User-defined nodes	671
Supported code pages	671
WebSphere MQ connections	699
User database connections	700
User database DBCS restrictions and UNICODE support	702
Data integrity within message flows.	702
Validation properties for messages in the MRM domain	703
Parsing on demand	706
Exception list structure	707
Configurable message flow properties	715
Message flow porting considerations	716
Message flow accounting and statistics data	717
Coordinated message flows.	734
Element definitions for message parsers	735
Message mappings	748
XML constructs.	762
Data sources on z/OS	778

ESQL reference 779

Syntax diagrams: available types	780
ESQL data types in message flows	780
ESQL variables	791
ESQL field references.	792
ESQL operators.	798
ESQL statements	804
ESQL functions: reference material, organized by function type	889
Broker properties accessible from ESQL and Java	983
Special characters, case sensitivity, and comments in ESQL	986
ESQL reserved keywords	988
ESQL non-reserved keywords	988
Example message	991

Flow application debugger 993

Java Debugger	993
Flow debugger shortcuts	993
Flow debugger icons and symbols	994

Part 5. Appendixes 997

Appendix. Notices 999

Trademarks	1001
----------------------	------

Index 1003

About this topic collection

This PDF has been created from the WebSphere Message Broker Version 6.0 (Tooling Version 6.0.0.1 update, March 2006) information center topics. Always refer to the WebSphere Message Broker online information center to access the most current information. The information center is periodically updated on the document update site and this PDF and others that you can download from that Web site might not contain the most current information.

The topic content included in the PDF does not include the "Related Links" sections provided in the online topics. Links within the topic content itself are included, but are active only if they link to another topic in the same PDF collection. Links to topics outside this topic collection are also shown, but these attempt to link to a PDF that is called after the topic identifier (for example, ac12340_.pdf) and therefore fail. Use the online information to navigate freely between topics.

Feedback: do not provide feedback on this PDF. Refer to the online information to ensure that you have access to the most current information, and use the Feedback link that appears at the end of each topic to report any errors or suggestions for improvement. Using the Feedback link provides precise information about the location of your comment.

The content of these topics is created for viewing online; you might find that the formatting and presentation of some figures, tables, examples, and so on are not optimized for the printed page. Text highlighting might also have a different appearance.

Part 1. Developing message flows

Developing message flows	3	Connecting nodes	140
Message flows overview	3	Removing a connection	142
Message flow projects	4	Adding a bend point	143
Message flow nodes	4	Removing a bend point	143
Message flow version and keywords	7	Aligning and arranging nodes	144
Connections	8	Developing ESQL	145
The message tree	9	ESQL overview	146
Parsers	26	Managing ESQL files	156
Properties	49	Writing ESQL	168
Message flow transactions	51	Developing Java	285
Broker schemas	52	Managing Java Files	286
Message flow accounting and statistics data	54	Writing Java	288
Designing a message flow	59	Developing message mappings	302
Deciding which nodes to use	60	Message mappings overview	303
Using more than one input node	68	Creating message mappings	306
Defining input message characteristics	68	Message mapping scenarios	326
Using nodes for decision making	69	Defining promoted properties	365
Using subflows	71	Promoting a property	365
Optimizing message flow response times	73	Renaming a promoted property	369
System considerations for message flow development	75	Removing a promoted property	370
Creating destination lists	76	Converging multiple properties	371
Using WebSphere MQ cluster queues for input and output	77	Collecting message flow accounting and statistics data	374
Using WebSphere MQ shared queues for input and output (z/OS)	78	Starting to collect message flow accounting and statistics data	374
Validating messages	79	Stopping message flow accounting and statistics data collection	377
Viewing the logical message tree in trace output	81	Viewing message flow accounting and statistics data collection parameters	378
Accessing databases from message flows	84	Modifying message flow accounting and statistics data collection parameters	379
Accessing databases from ESQL	86	Resetting message flow accounting and statistics archive data	380
The Transactional model	88	Developing user exits	380
Message flow aggregation	90	Developing a user exit	383
Data conversion	92	Deploying a user exit	383
Configuring coordinated message flows	93	Configuring aggregation flows	384
Configuring message flows for data conversion	107	Creating the aggregation fan-out flow	385
Ensuring that messages are not lost	109	Creating the aggregation fan-in flow	389
Handling errors in message flows	111	Associating fan-out and fan-in aggregation flows	393
Managing message flows	122	Setting timeouts for aggregation	394
Creating a message flow project	123	Using multiple AggregateControl nodes	395
Deleting a message flow project	124	Correlating input request and output response aggregation messages	396
Creating a broker schema	125	Using control messages in aggregation flows	397
Creating a message flow	126	Handling exceptions in aggregation flows	400
Opening an existing message flow	127	Configuring timeout flows	402
Copying a message flow using copy	127	Timeout request message	402
Renaming a message flow	128	Sending a message after a timed interval	404
Moving a message flow	129	Sending a message multiple times after a specified start time	405
Deleting a message flow	130	Automatically generating messages to drive a flow	406
Deleting a broker schema	131	Using an MQGet node in a request-response flow	407
Object version and keyword	131	Introduction	407
Saving a message flow	132		
Defining message flow content	135		
Adding a node	135		
Adding a subflow	136		
Renaming a node	137		
Configuring a node	138		
Removing a node	139		

How the LocalEnvironment is used	408
How the MQMD for the MQGet call is constructed	409
How the output message tree is constructed . .	410
Message tree examples	411

Developing message flows

There are many tasks involved in creating and maintaining message flows:

- “Designing a message flow” on page 59
- “Creating a message flow project” on page 123
- “Deleting a message flow project” on page 124
- “Creating a broker schema” on page 125
- “Creating a message flow” on page 126
- “Opening an existing message flow” on page 127
- “Defining message flow content” on page 135
- “Saving a message flow” on page 132
- “Copying a message flow using copy” on page 127
- “Renaming a message flow” on page 128
- “Moving a message flow” on page 129
- “Deleting a message flow” on page 130
- “Deleting a broker schema” on page 131
- “Developing message mappings” on page 302
- “Developing ESQL” on page 145
- “Developing Java” on page 285
- “Defining promoted properties” on page 365
- “Collecting message flow accounting and statistics data” on page 374
- “Configuring aggregation flows” on page 384
- “Configuring timeout flows” on page 402

The workbench provides a set of toolbar icons that invoke wizards that you can use to create any of the resources associated with message flows, for example, message flow projects and ESQL files. Hover your mouse pointer over each icon to see its function.

The workbench lets you open resource files with other editors. Use only the workbench message flow editor to work with message flow files, because this editor correctly validates all changes that you make to these files when you save the message flow.

When you have completed developing your message flow, deploy it to a broker to start its execution.

Tip: You can debug your message flow using the flow debugger.

Message flows overview

A message flow is a sequence of processing steps that execute in the broker when an input message is received.

You define a message flow in the workbench by including a number of message flow nodes, each of which represents a set of actions that define a processing step. The connections in the flow determine which processing steps are carried out, in which order, and under which conditions. A message flow must include an input node that provides the source of the messages that are processed. You must then deploy the message flow to a broker for execution.

When you want to exchange messages between multiple applications, you might find that the applications do not understand or expect messages in exactly the

same format. You might need to provide some processing between the sending and receiving applications that ensures that both can continue to work unchanged, but can exchange messages successfully.

You define the processing that is required when you create and configure a message flow. The way that you do this determines what actions are performed on a message when it is received, and the order in which the actions are completed.

You can create a message flow using the built-in nodes, nodes that you or a vendor have created (user-defined nodes), or other message flows (known as subflows). When you want to invoke a message flow to process messages, you deploy it to a broker, where it is executed within an execution group.

The following topics describe the concepts that you need to understand to design, create, and configure a message flow and its associated resources:

- Projects
- Nodes
- Version and keywords
- “Connections” on page 8
- “Parsers” on page 26
- “Logical tree structure” on page 14
- “Properties” on page 49
- Accounting and statistics data
- “Message flow transactions” on page 51
- Aggregation
- “Data conversion” on page 92
- “Message flows, ESQL, and mappings” on page 6
- “Broker schemas” on page 52
- “Message mappings overview” on page 303
- “ESQL overview” on page 146

Message flow projects

A message flow project is a specialized container in which you create and maintain all the resources associated with one or more message flows.

You can create a message flow project to contain a single message flow and its resources, or you can group together related message flows and resources in a single message flow project to provide an organizational structure to your message flow resources.

Message flow project resources are created as files, and are displayed within the project in the Navigator view. These resources define the content of the message flow, and additional objects that contain detailed configuration information, in the form of ESQL modules and mappings, for one or more nodes within the message flow.

Import one of the samples from the Samples Gallery to see how the sample’s message flow resources are stored in a Message Flow project. If the sample has a message set, its message set resources are stored in a Message Set project. For example, import the Video Rental sample or the Comma Separated Value (CSV) sample; both of these samples have a Message Flow project and a Message Set project.

Message flow nodes

A message flow node is a processing step in a message flow.

It receives a message, performs a set of actions against the message, and optionally passes the message on to the next node in the message flow. A message flow node can be a built-in node, a user-defined node, or a subflow node.

A message flow node has a fixed number of input and output points known as terminals. You can make connections between the terminals to define the routes that a message can take through a message flow.

Built-in node

A built-in node is a message flow node that is supplied by WebSphere Message Broker. The built-in nodes provide input and output, manipulation and transformation, decision making, collating requests, and error handling and reporting functions.

For information on all the built-in nodes supplied by WebSphere Message Broker, see “Built-in nodes” on page 482.

User-defined node

A user-defined node is an extension to the broker that provides a new message flow node in addition to those supplied with the product. It must be written to the user-defined node API provided by WebSphere Message Broker for both C and Java languages. The User-defined Extension sample demonstrates how you can write your own nodes in both C and Java languages.

Subflow

A subflow is a directed graph that is composed of message flow nodes and connectors and is designed to be embedded in a message flow or in another subflow. A subflow must include at least one Input node or one Output node. A subflow can be executed by a broker only as part of the message flow in which it is embedded, and therefore cannot be independently deployed.

A message is received by an Input node and processed according to the definition of the subflow. That might include being stored through a Warehouse node, or delivered to another message target, for example through an MQOutput node. If required, the message can be passed through an Output node back to the main flow for further processing.

The subflow, when it is embedded in a main flow, is represented by a subflow node, which has a unique icon. The icon is displayed with the correct number of terminals to represent the Input and Output nodes that you have included in the subflow definition.

The most common use of a subflow is to provide processing that is required in many places within a message flow, or is to be shared between several message flows. For example, you might code some error processing in a subflow, or create a subflow to provide an audit trail (storing the entire message and writing a trace entry).

The use of subflows is demonstrated in the Error Handler sample and the Coordinated Request Reply sample. The Error Handler sample uses a subflow to trap information about errors and store the information in a database. The Coordinated Request Reply sample uses a subflow to encapsulate the storage of the ReplyToQ and ReplyToQMgr values in a WebSphere MQ message so the processing logic can be reused in other message flows and to allow alternative implementations to be substituted.

A node does not always produce an output message for every output terminal: often it produces one output for a single terminal based on the message received

or the result of the operation of the node. For example, a Filter node typically sends a message on either the true terminal or the false terminal, but not both.

If more than one terminal is connected, the node sends the output message on each terminal, but sends on the next terminal only when the processing has completed for the current terminal. Updates to a message are never propagated to previously-executed nodes, only to nodes following the node in which the update has been made. The order in which the message is propagated to the different output terminals is determined by the broker; you cannot change this order. The only exception to this rule is the FlowOrder node, in which the terminals indicate the order in which the message is propagated to each.

The message flow can accept a new message for processing only when all paths through the message flow (that is, all connected nodes from all output terminals) have been completed.

The Airline Reservations sample uses Environment variables in the XML_Reservation sample to store information that has been taken from a database table and to pass that information to a node downstream in the message flow.

Message flows, ESQL, and mappings

A message flow represents the set of actions performed on a message when it is received and processed by a broker. The content and behavior of a message flow is defined by a set of files that you create when you complete your definition and configuration of the message flow content and structure:

- The message flow definition file `<message_flow_name>.msgflow`. This is a required file and is created automatically for you. It contains details about the message flow characteristics and contents (for example, what nodes it includes, its promoted properties, and so on).
- The ESQL resources file `<message_flow_name>.esql`. This file is required only if your message flow includes one or more of the nodes that must be customized using ESQL modules. You can create this file yourself, or you can cause it to be created for you by requesting specific actions against a node.

You can customize the following built-in nodes by creating free-form ESQL statements that use the built-in ESQL statements and functions, and your own user-defined functions:

- Compute
- Database
- Filter

- The message mappings file `<message_flow_name><_nodename>.msgmap`. This file is required only if your message flow contains one or more of the nodes that must be customized using mappings. You can create this file yourself, or you can cause it to be created for you by requesting specific actions against a node. A different file is required for each node in the message flow that uses the Message Mapping editor.

You can customize the following built-in nodes by specifying how input values map to output values:

Node	Usage
"DataDelete node" on page 504	Use this node to delete one or more rows from a database table without creating an output message.

Node	Usage
“DataInsert node” on page 507	Use this node to insert one or more rows in a database table without creating an output message.
“DataUpdate node” on page 510	Use this node to update one or more rows in a database table without creating an output message.
“Extract node” on page 513	Use this node to create a new output message that contains a subset of the contents of the input message. Use the Extract node only if no database is involved in the map
“Mapping node” on page 567	Use this node to construct a new output message and populate it with information that is new, modified from the input message, or taken from a database. Use the Mapping node only if no database is involved in the map.
“Warehouse node” on page 661	Use this node to store all or part of a message in a database table without creating an output message.

You can use built-in ESQL functions and statements to define message mappings, and you can use your own ESQL functions.

Message flow version and keywords

When you are developing a message flow, you can define the version of the message flow as well as any other key information that you need to be associated to it. After the message flow has been deployed, the Configuration Manager can be used to display the properties of the message flow. This includes the deployment and modification dates and times (the default information that is displayed) as well as any additional version or keyword information that you have set.

You can define information to give details on the message flow that has been deployed, therefore you can check that it is the message flow that you expect.

Version

You can set the version of the message flow in the Version property. This is in the Properties dialog.

You can also define a default message flow version in the Default version tag of the message flow preferences. All new message flows that are created after this has been set have this default applied to the Version property at the message flow level.

Keywords

Keywords are extracted from the compiled message flow (the .cmf file) rather than the message flow source (the .msgflow file). Not all of the source properties are added to the compiled file. Therefore, add message flow keywords in only these places:

- The label property of a passthrough node
- ESQL comments or string literals
- The Long Description property of the message flow

Any keywords that you define need to follow certain rules to ensure that the information can be parsed. The following is an example of what you can define in the Long Description property:

```
$MQSI Author=John Smith MQSI$  
$MQSI Subflow 1 Version=v1.3.2 MQSI$
```

The information the Configuration Manager will show is:

Message flow name	
Deployment Time	28-Aug-2004 15:04
Modification Time	28-Aug-2004 14:27
Version	v1.0
Author	John Smith
Subflow 1 Version	v1.3.2

In this display the version information has also been defined using the Version property of the object. If the version information has not been defined using the property, it is omitted from this display.

If message flows contain subflows, you can embed keywords in each subflow.

Connections

A connection is an entity that connects an output terminal of one message flow node to the input terminal of another. It represents the flow of control and data between two message flow nodes.

The connections of the message flow, represented by black lines within the message flow editor view, determine the path that a message takes through the message flow. You can add bend points to the connection to alter the way in which it is displayed.

See “Bend points” for a description of bend points. See “Terminals” for a description of terminals.

Bend points

A bend point is a point that is introduced in a connection between two message flow nodes at which the line that represents the connection changes direction.

Use bend points to change the visual path of a connection to display node alignment and processing logic more clearly and effectively. Bend points have no effect on the behavior of the message flow; they are visual modifications only.

A connection is initially made as a straight line between the two connected nodes or brokers. Use bend points to move the representation of the connection, without moving its start and end points.

Terminals

A terminal is the point at which one node in a message flow is connected to another node.

Use terminals to control the route that a message takes, depending whether the operation performed by a node on that message is successful. Terminals are wired to other node terminals using message flow node connections to indicate the flow of control.

Every built-in node has a number of terminals to which you can connect other nodes. Input nodes (for example, MQInput) do not have in terminals; all other nodes have at least one in terminal through which to receive messages to be processed. Most built-in nodes have failure terminals that you can use to manage the handling of errors in the message flow. Most nodes have output terminals through which the message can flow to a subsequent node.

If you have any user-defined nodes, these might also have terminals that you can connect to other built-in or user-defined node terminals.

The message tree

A message tree is a structure that is created by one or more parsers when an input message bit stream is received by a message flow or by the action of a message flow node.

A message is used to describe:

- A set of business data exchanged by applications
- A set of elements arranged in a predefined structure
- A structured sequence of bytes

WebSphere Message Broker routes and manipulates messages after converting them into a logical tree. The process of conversion, called parsing, makes the content and structure of a message obvious, and simplifies later operations. After the message has been processed, the parser converts it back into a bit stream.

WebSphere Message Broker supplies a range of parsers to handle the many different messaging standards in use. In some cases WebSphere Message Broker will be able to parse your message automatically. Alternatively, the MRM provides a powerful method of modeling messages that you need to process.

After a message has been parsed, it can be processed in a message flow.

The logical tree contains identical contents to that of the message but is easier to manipulate within the message flow. The message flow nodes provide an interface to query, update or create the content within the tree.

How the message tree is populated

The message tree is initially populated by the input node of the message flow.

When the input node receives the input message, it creates the Properties tree (the first subtree of the message tree) and populates this with the node properties that you have configured in the message flow. It then examines the contents of the input message bit stream and creates the remainder of the message tree to reflect those contents. This process depends to some extent on the input node itself, which is governed by the transport across which the message is received:

WebSphere MQ Enterprise Transport, and WebSphere MQ Telemetry Transport protocols

If your application communicates with the broker across these protocols, and your message flow includes the corresponding MQInput, or

SCADAInput node, all messages that are received must start with a Message Queue Message Descriptor (MQMD) header. If a valid MQMD is not present at the start of the message, the message is rejected and no further processing takes place.

The input node first invokes the MQMD parser and creates the subtree for that header.

A message can have zero or more additional headers following the MQMD, and these are chained together, with the Format field of one header defining the format of the following header, up to and including the last header, which defines the format of the message body. If there is an MQRFH and MQRFH2 header in the chain, the name/value data in either of these two headers can also contain information about the format of the following data. If the value specified in Format is a recognized parser, this always takes precedence over the name/value data.

The broker invokes the appropriate parser to interpret each header, following the chain in the message. Each header is parsed independently. The fields within a single header are parsed in an order that is governed by the parser. You cannot predict or rely on the order chosen, but the order in which fields are parsed does not affect the order in which the fields appear within the header.

The broker ensures that the integrity of the headers that precede a message body is maintained. The format of each part of the message is defined, either by the Format field in the immediately preceding header (if the following part is a recognized WebSphere MQ format) or by the values set in the MQRFH or MQRFH2 header:

- The format of the first header is known, because this must be MQMD.
- The format of any subsequent header in the message is set in the Format field in the preceding header.
- The format of the body corresponds to the message domain and the parser that must be invoked for the message body (for example, XML). This information is set either in the MQRFH or MQRFH2 header, or in the message domain property of the input node that receives the message.

This process is repeated as many times as required by the number of headers that precede the message body. You do not have to populate these fields yourself; the broker handles this sequence for you.

The broker completes this process to ensure that Format fields in headers correctly identify each part of the message. If the broker does not do this, WebSphere MQ might be unable to deliver the message. Because the message body parser is not a recognized WebSphere MQ header format, the broker replaces this value in the last header's Format field with the value MQFMT_NONE. The original value in that field is stored in the Domain field within the MQRFH or MQRFH2 header to retain the information about the contents of the message body. If there is no MQRFH or MQRFH2, the information is stored in the Properties tree.

For example, if the MQRFH2 header immediately precedes the message body and its Format field is set to XML, indicating that the message body must be parsed by the generic XML parser, the MQRFH2 Domain field is set to XML, and its Format field reset to MQFMT_NONE.

These actions might result in information explicitly stored by an ESQL expression being replaced by the broker.

When all the headers have been parsed, and the corresponding subtrees created within the message tree, the input node associates the specified parser with the message body. You must specify the parser that is to be associated with the message body content. You can do this either in a header in the message, for example the <mcd> folder within the MQRFH2 header (generally recommended), or in the input node properties (recommended if the message does not include headers). The input node makes the association as described below:

- If the message has an MQRFH or MQRFH2 header, the domain that is identified in the header (either in Format or the name/value data) determines the parser that is associated with this message.

The SCADAInput node creates WebSphere MQ format messages with MQRFH2 headers from the input messages that the listener receives on the TCP/IP port.

- If the message does not have an MQRFH or MQRFH2 header, or the header does not identify the domain, but the properties of the input node, stored in the Properties tree, indicate the domain of the message, the parser specified by the input node property is used. You can specify a user-defined parser.
- If the message domain cannot be identified by header values or by properties of the input node, the message is handled as a binary object (BLOB). The BLOB parser is associated with the message. A BLOB can be interpreted as a string of hexadecimal characters, and can be modified or examined in the message flow by specifying the location of the subset of the string.

The message body is not parsed for performance reasons; it is possible that the configuration of the message flow does not require the message body to be parsed. The body is parsed only when a reference is made to its contents during the message flow.

For example, the message body is parsed when you refer to `Root.XML.Field` (or `InputRoot.XML.Field` in the Compute node) or `Root.MRM.Field`. Depending on paths taken in the message flow, this parse could take place at different points. This parse when first needed approach is also referred to as partial parsing, and in normal processing does not affect the logic of a message flow. However, there are some implications for error handling scenarios, as described in “Handling errors in message flows” on page 111.

If you want a message flow to accept messages from more than one message domain, you can include an MQRFH2 header in your message from which the input nodes extract the message domain and related message definition information (set, type, and format).

If you set up the message headers or the input node properties to identify a user-defined parser, the way in which it interprets the message and constructs the logical tree might differ from that described here.

WebSphere MQ Multicast Transport, WebSphere MQ Real-time Transport, and WebSphere MQ Web Services Transport protocols

If your application communicates with the broker across these supported protocols, and your message flow includes the corresponding Real-timeInput, HTTPInput, or HTTPRequest nodes, messages that are received are not required to include a particular header. Your applications can include the MQRFH2 header to provide message-related information, for example about publications and subscriptions. If recognized headers

are included, the input node invokes the appropriate parsers to interpret the headers and to build the relevant parts of the message tree, as described for the other supported protocols.

If there are no headers, or these headers do not specify the parser for the message body, you must set the input node properties to define the message body parser. If you do not do so, the message is treated as a BLOB. You can specify a user-defined parser.

The specified parser is associated with the message body by the input node (in the same way as it is for the WebSphere MQ Enterprise Transport, WebSphere MQ Mobile Transport, and WebSphere MQ Telemetry Transport protocols) and the message body is not parsed.

If you set up the message headers or the input node properties to identify a user-defined parser, the way in which it interprets the message and constructs the logical tree might differ from that described here.

All other protocols

If you want your message flow to accept messages from a transport protocol for which WebSphere Message Broker does not provide built-in support, or you want it to provide some specific processing on receipt of a message, use either the Java or the C language programming interface to create a new user-defined input node.

This interface does not automatically generate a Properties subtree for a message (this subtree is discussed in “Message tree” on page 15). It is not a requirement for a message to have a Properties subtree, although you might find it useful to create one to provide a consistent message tree structure regardless of input node. If you want to create a Properties subtree within the message tree, and you are using a user-defined input node, you must do this yourself.

If you need to process messages that do not conform to any of the defined message domains, you can use the C language programming interface to create a new user-defined parser.

Refer to the node interface to understand how it uses parsers, and whether you can configure it to modify its behavior. If the node uses a user-defined parser, the tree structure created for the message might differ slightly from that created for built-in parsers. A user-defined input node can parse an input message completely, or it can participate in partial parsing in which the message body is parsed only when it is required.

You can also create your own output and message processing nodes in C or Java.

Tree contents after an exception:

The contents of the message tree are updated if an exception is raised.

If no exception occurs while processing the message, the tree structure and content received by an individual node is determined by the action of previous nodes in the flow.

If an exception occurs in the message flow, the content of the four trees depends on the following factors:

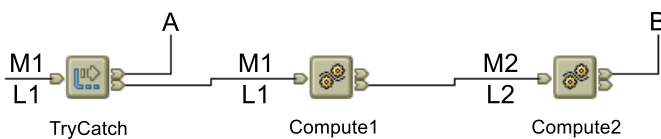
- If the exception is returned to the input node, and the input node catch terminal is not connected, the trees are discarded. If the message is within a transaction, it

is returned to the input queue for further processing. When the message is processed again, a new tree structure is created. If the message is not within a transaction, it is discarded.

- If the exception is returned to the input node and the catch terminal is connected, the Message and LocalEnvironment trees that were created originally by the input node, and propagated through the out terminal, are restored, and any updates that you made to their content in the nodes that followed the input node are lost. The Environment tree is not restored, and its contents are preserved. If the nodes following the input node include a Compute node that creates a new LocalEnvironment or Message tree, those trees are lost. The ExceptionList tree reflects the one or more exceptions that have been recorded.
- If the exception is caught within the message flow by a TryCatch node, the Message and LocalEnvironment trees that were previously propagated through the try terminal of the TryCatch node are restored and propagated through the catch terminal. Any updates that you made to their content in the nodes that followed the TryCatch node are lost. The Environment tree is not restored, and its contents are preserved. If the nodes following the TryCatch node include a Compute node that creates a new LocalEnvironment or Message tree, those trees are lost. The ExceptionList tree reflects the one or more exceptions that have been recorded.

Exception handling paths in a message flow: Exception handling paths start at a failure terminal (most message processing nodes have these), the catch terminal of an input node, a TryCatch node, or an AggregateReply node, but are no different in principle from a normal message flow path. Such a flow consists of a sequence of nodes connected together by the designer of the message flow. The exception handling paths differ in the kind of processing that they do to record or react to the exception. For example, they might examine the exception list to determine the nature of the error, and take appropriate action or log data from the message or exception.

The LocalEnvironment and message tree that are propagated to the exception handling message flow path are those at the start of the exception path, not those at the point when the exception is thrown. The figure below illustrates this point:



- A message (M1) and LocalEnvironment (L1) are being processed by a message flow. They are passed through the TryCatch node to Compute1.
- Compute1 updates the message and LocalEnvironment and propagates a new message (M2) and LocalEnvironment (L2) to the next node, Compute2.
- An exception is thrown in Compute2. If the failure terminal of Compute2 is not connected (point B), the exception is propagated back to the TryCatch node, but the message and LocalEnvironment are not. The exception handling path starting at point A has access to the first message and LocalEnvironment, M1 and L1. The Environment tree is also available and retains the content it had when the exception occurred.

- If the failure terminal of Compute2 is connected (point **B**), the message and LocalEnvironment M2 and L2 are propagated to the node connected to that failure terminal. The Environment tree is also available and retains the content it had when the exception occurred.

Logical tree structure

The logical tree structure is the internal (broker) representation of a message.

When a message arrives at a broker, it is received by an input node that you have configured in a message flow. Before the message can be processed by the message flow, it must be interpreted by one or more parsers that create a logical tree representation from the bit stream of the message data.

The tree format contains identical content to the bit stream from which it is created, but it is easier to manipulate within the message flow. Many of the built-in message flow nodes provide an interface for you to query and update message content within the tree, and perform other actions against messages and databases to help you to provide the required function in each node.

Three interfaces are provided:

- *ESQL*, a programming language that you can code in the Compute, Database, and Filter nodes.
- *Java*, a programming language that you can code in the JavaCompute node.
- *Mappings*, a method of achieving transformation from input to output structures, available in the DataDelete, DataInsert, DataUpdate, Extract, Mapping, and Warehouse nodes.

The tree structure created by the parsers is largely independent of any message format (for example, XML). The exception to this is the subtree that is created as part of the message tree and that represents the message body. This subtree is message dependent, and its content is not further described.

The input node creates the logical tree, which consists of four subtrees:

- “Message tree” on page 15
- “Environment tree” on page 17
- “LocalEnvironment tree” on page 18
- “ExceptionList tree” on page 21

The first of these trees is populated with the contents of the input message bit stream, as described in “How the message tree is populated” on page 9: the remaining three are initially empty.

Each of the four trees created has a root element (with a name that is specific to each tree). Each tree is made up of a number of discrete pieces of information called elements. The root element has no parent and no siblings (siblings are elements that share a single parent). The root is parent to a number of child elements. Each child must have a parent, can have zero or more siblings, and can have zero or more children.

The four trees are created for both built-in and user-defined input nodes and parsers.

The input node passes the logical tree structure that it has created to subsequent message processing nodes in the message flow:

- All message processing nodes can read the four trees.

- You can code ESQL in the Database and Filter nodes, or use mappings in the nodes that support that interface to modify the Environment and LocalEnvironment trees only.
- The Compute node differs from other nodes in that it has both an input tree and at least one output tree. Configure the Compute node to determine which trees are included in the output message, with the exception of the Environment tree, which is always retained from input tree to output tree.

To determine which of the other trees are included, you must specify a value for the *Compute mode* property of the node (displayed on the Advanced tab). The default action is for only the message to be created. You can specify any combination of message, LocalEnvironment, and ExceptionList trees to be created in the output tree.

If you want the output tree to contain a complete copy of the input message tree, you can code a single ESQL SET statement to make the copy. If you want the output message to contain a subset of the input message tree, code ESQL to copy those parts that you want. In both cases, your choice of *Compute mode* must include Message.

If you want the output tree to contain all or part of the input LocalEnvironment or ExceptionList tree contents, code the appropriate ESQL to copy information you want to retain in that tree. Your choice of *Compute mode* must include LocalEnvironment, or Exception, or both.

You can also code ESQL to populate the output message, Environment, LocalEnvironment, or ExceptionList tree with information that is not copied from the input tree. For example, you can retrieve data from a database, or calculate content from the input message data.

Message tree:

The message tree is a part of the logical message tree in which the broker stores its internal representation of the message body.

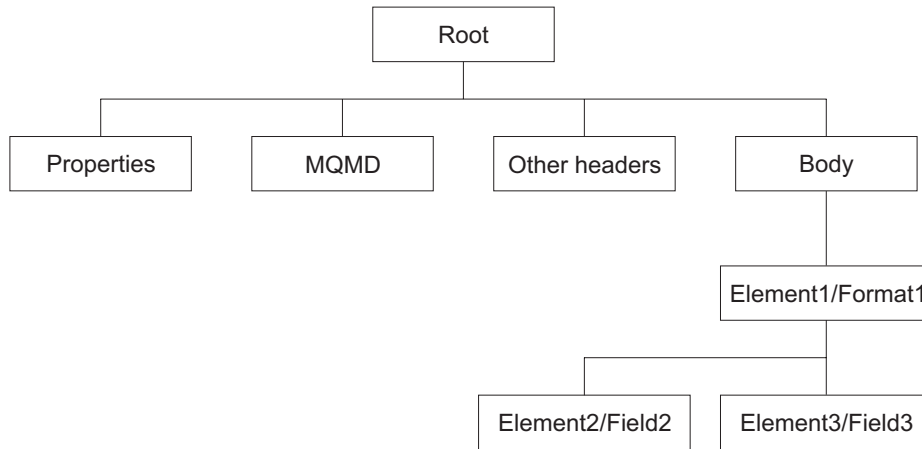
The root of a message tree is called Root. The message tree is always present, and is passed from node to node within a single instance of a message flow.

The message tree includes all the headers that are present in the message, in addition to the message body. It also includes the properties subtree (described in “Parsers” on page 26), if that is created by the parser. If a supplied parser has created the message tree, the element representing the properties subtree is followed by none or more headers.

If the message has been received across the WebSphere MQ Enterprise Transport, WebSphere MQ Mobile Transport, or WebSphere MQ Telemetry Transport, the first header (the second element) must be the MQMD. Any additional headers included in the message appear in the tree in the same order as in the message. The last element beneath the root of the message tree is always the message body.

If a user-defined parser has created the message tree, the Properties tree, if present, is followed by the message body.

The message tree structure is shown below. If the input message is not a WebSphere MQ message, the headers shown might not be present. If the parser that created this tree is a user-defined parser, the Properties tree might not be present.



The Body tree, a structure of child elements (described below) that is determined by the parser, represents the message content (data). The message body might be:

- MRM, if you have modeled the message
- XML, if the message is self-defining
- A value that identifies a user-defined parser

Each element within the parsed tree is one of three types:

Name element

A name element has a string associated with it, which is the name of the element. An example of a name element is `XMLElement`, described in “XML element” on page 767.

Value element

A value element has a value associated with it. An example of a value element is `XMLContent`, described in “XML content” on page 767.

Name-value element

A name-value element is an optimization of the case where a name element contains only a value element and nothing else. The element contains both a name and a value. An example of a name-value element is `XMLAttribute`, described in “XML attribute” on page 766.

For information about how the message tree is populated, see “How the message tree is populated” on page 9.

Properties folder:

The properties folder is the first element of the message tree and holds information about the characteristics of the message.

The root of the properties folder is called Properties. It is the first element under Root. All message trees generated by the built-in parsers include a properties folder for the message. If you create your own user-defined parser, you can choose whether the parser creates a properties folder. However, for consistency, you are recommended to include this action in the user-defined parser.

The properties folder is created and inserted in the tree following all the headers but preceding the message data. It contains a set of standard properties that you can manipulate in the message flow nodes in the same way as any other property.

The majority of these fields map to fields in the supported WebSphere MQ headers, if present, and are passed to the appropriate parser when a message is delivered from one node to another.

For example, the MQRFH2 header contains information about the message set, type, and format. These values are stored in the properties folder as MessageSet, MessageType, and MessageFormat. To access these values using ESQL within the message processing nodes, refer to these values in the properties folder; do not refer directly to the fields in the headers from which they are derived.

If the message is converted to a bit stream, for example in an output node, any properties remaining solely in the properties folder (that is, not in any header in the output messages) are not included in any part of the output message.

The Properties parser ensures that the values in the header fields match the values in the properties folder on input to, and output from, every node. On exit from a node, the Properties parser invokes each header parser with the values that it currently contains. It then requests values back from the header parser and updates its own values. If you have coded ESQL in the node that updates values either in the properties folder, or in the header, or both, these values always match when the tree is passed on from that node. However, if you have updated a field in both the properties folder and the header with different values, the value that you set in the header is overwritten by the value that you set in the properties folder.

When the message flow processing is complete, the properties folder is discarded.

Environment tree:

The Environment tree is a part of the logical message tree in which you can store information while the message passes through the message flow.

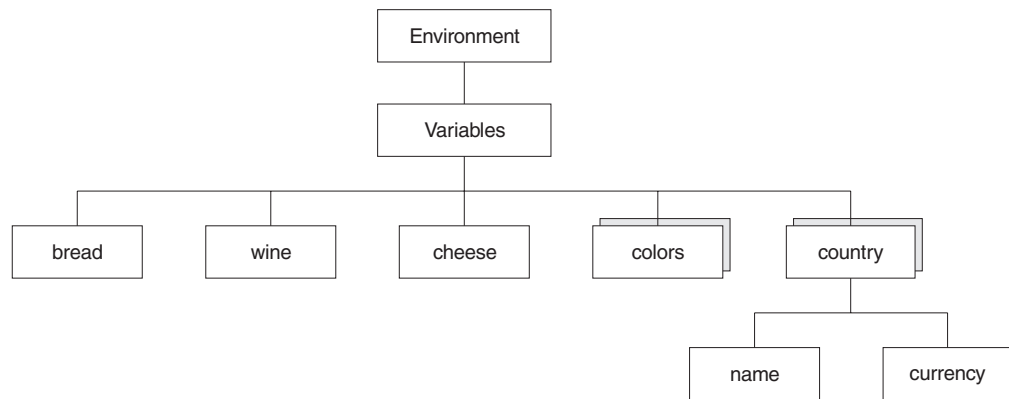
The root of the Environment tree is called Environment. This tree is always present in the input message; an empty Environment tree is created when a message is received and parsed by the input node. You can use this tree as you choose, and create both its content and structure.

There is one situation in which WebSphere Message Broker refers to (but never creates) a field in this tree. If you have requested data collection for message flow accounting and statistics, and have indicated that accounting origin basic support is required, the broker checks for the existence of the field Environment.Broker.AccountingOrigin. If the field exists, the broker uses its value to set the accounting origin for the current data record. For further information about the use of this field, see “Setting message flow accounting and statistics accounting origin” on page 376. (Contrast this with the “LocalEnvironment tree” on page 18, which the broker uses in several situations.)

The Environment tree differs from the LocalEnvironment tree in that a single instance of it is maintained throughout the message flow. If you include a Compute node in your message flow, you do not have to specify whether you want the Environment tree to be included in the output message. This happens automatically, and the entire contents of the input Environment tree are retained in the output Environment tree, subject to any modifications that you make using ESQL in the node. Any changes that you make are available to subsequent nodes in the message flow, and to previous nodes if the message flows back (for example, to a FlowOrder or TryCatch node).

You are recommended to create information in the Environment tree within a subtree called Variables (although this is not enforced).

An example of an Environment tree is shown below.



You could use the following ESQL statements to create the content shown above.

```
SET Environment.Variables =  
    ROW('granary' AS bread, 'riesling' AS wine, 'stilton' AS cheese);  
SET Environment.Variables.Colors[] =  
    LIST{'yellow', 'green', 'blue', 'red', 'black'};  
SET Environment.Variables.Country[] = LIST{ROW('UK' AS name, 'pound' AS currency),  
    ROW('USA' AS name, 'dollar' AS currency)};
```

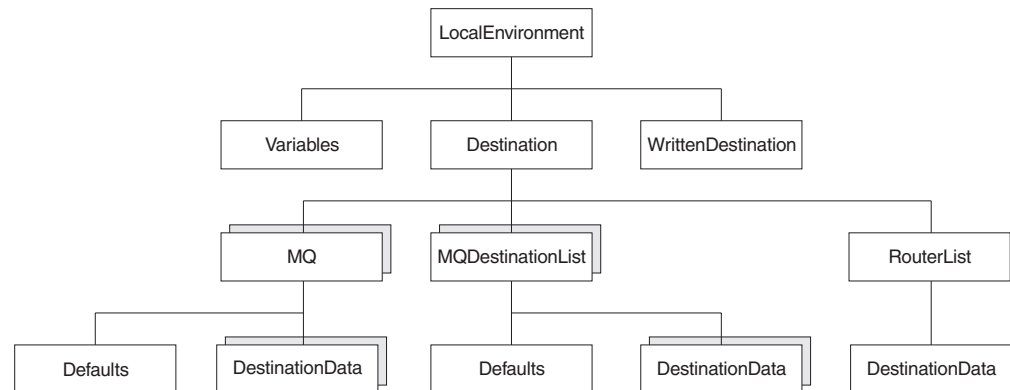
When the message flow processing is complete, the Environment tree is discarded.

LocalEnvironment tree: The LocalEnvironment tree is a part of the logical message tree in which you can store information while the message flow processes the message.

The root of the LocalEnvironment tree is called LocalEnvironment. This tree is always present in the input message: an empty LocalEnvironment tree is created when a message is received by the input node.

Use the LocalEnvironment tree to store variables that can be referred to and updated by message processing nodes that occur later in the message flow. You can also use the LocalEnvironment tree to define destinations (that are internal and external to the message flow) to which a message is sent. WebSphere Message Broker also stores information in LocalEnvironment in some circumstances, and references it to access values that you might have set for destinations. (Contrast this to the “Environment tree” on page 17, which the broker refers to in one situation only.)

The figure below shows the LocalEnvironment tree structure. The children of Destination are protocol-dependent. The example below shows the structure for a WebSphere MQ message. If the message is parsed by another built-in parser, or by a user-defined parser, the tree structure below the Destination element has different content than that shown in the figure.



In the tree structure shown above, LocalEnvironment has three children:

Variables

This subtree is optional. If you create local environment variables, you are recommended to store them in a subtree called Variables. It provides a work area that you can use to pass information between nodes. This subtree is never inspected or modified by any supplied node.

Variables in the local environment can be changed by any subsequent message processing node, and persist until the message flow goes out of scope and the node that created it has completed its work and returns control to the previous node

The variables in this subtree are only persistent within a single instance of a message flow. If you have multiple instances of a message passing through the message flow, and need to pass information between them, you must use an external database.

Destination

This subtree consists of a number of children that indicate the transport types to which the message is directed (the Transport identifiers), or the target Label nodes that are used by a RouteToLabel node.

- Transport information

Transport information is used by some input and output nodes.

If the message flow starts with an HTTPInput node, a single name element HTTP is added to Destination. The element HTTP.RequestIdentifier is created and initialized so that it can be used by an HTTPReply node. You can also create other fields in the HTTP structure for use by the HTTPRequest node, for example the URL of the service to which the request is sent. The topic for each node contains more information about the contents of Destination for the WebSphere MQ Web Services Transport protocol.

If the message flow includes an MQOutput node, each element is a single name element, MQDestinationList. If more than one element exists, each is processed sequentially by the node.

If you have included a user-defined output node in the message flow, the contents of Destination (if supported) are defined by that node.

You can configure output nodes to examine the list of destinations and send the message to those destinations, by setting the property *Destination Mode* to Destination List. If you do so, you must create this subtree and its contents to define those destinations, giving it the name Destination. If you do not do so, the output node cannot deliver the messages.

If you prefer, you can configure the output node to send messages to a single fixed destination, by setting the property *Destination Mode to Queue Name* or *Reply To Queue*. If you select either of these fixed options, the destination list has no effect on broker operations and you do not have to create this subtree.

You can construct the MQ element to contain a single optional Defaults element. The Defaults element, if created, must be the first child and must contain a set of name-value elements that give default values for the message destination and its PUT options for that parent.

You can also create a number of elements called DestinationData within MQ. Each of these can be set up with a set of name-value elements that defines a message destination and its PUT options.

The set of elements that define a destination is described in “Data types for elements in the DestinationData subtree” on page 736.

The content of each instance of DestinationData is the same as the content of Defaults for each protocol, and can be used to override the default values in Defaults. You can set up Defaults to contain values that are common to all destinations, and set only the unique values in each DestinationData subtree. If you do not set a value either in DestinationData or Defaults then the value that you have set for the corresponding node property is used. Similarly, if you specify a field name or value with the wrong spelling or case, it is ignored, and the value that you have set for the corresponding node property is used.

The information that you insert into DestinationData depends on the characteristic of the corresponding node property: this is described in “Accessing the LocalEnvironment tree” on page 192.

- Routing information

The child of Destination is RouterList. It has a single child element called DestinationData, which has a single entry called labelName. If you are using a dynamic routing scenario involving the RouteToLabel and Label nodes, you must set up the Destination subtree with a RouterList that contains the reference labels.

WrittenDestination

This subtree contains the addresses to which the message has been written. Its name is fixed. It is created by the message flow when a message is propagated through the out terminal of an output node. It includes transport specific information (for example, if the output message has been put to a WebSphere MQ queue, it includes the queue manager and queue names). If the out terminal of the output node is not connected to another node, this subtree is not created.

The topic for each output node contains more information about the contents of WrittenDestination for WebSphere MQ and WebSphere MQ Everyplace.

When the message flow processing is complete, the LocalEnvironment tree is discarded.

The Airline Reservations sample and the Message Routing sample demonstrate how to use the LocalEnvironment to dynamically route messages based on the destination list. The User-defined Extension sample uses the LocalEnvironment to store information that is later added to the output message that is created by the message flow.

ExceptionList tree:

The ExceptionList tree is a part of the logical message tree in which the message flow writes information about exceptions that occur when a message is processed.

The root of the ExceptionList tree is called ExceptionList, and the tree consists of a set of zero or more exception descriptions. The ExceptionList tree is populated by the message flow if an exception occurs. If no exception conditions occur during message flow processing, the exception list associated with that message consists of a root element only. This is, in effect, an empty list of exceptions.

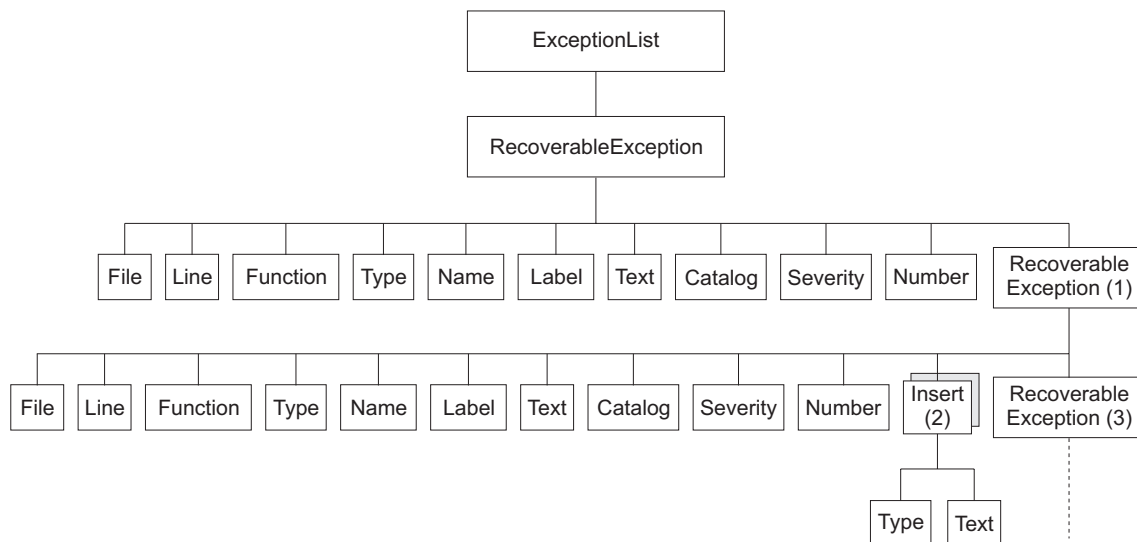
The ExceptionList tree can be accessed by other nodes within the message flow that receive the message after the exception has occurred. You can modify its contents only in the Compute node.

If an exception condition occurs, message processing is suspended and an exception is thrown. Control is passed back to a higher level, that is, an enclosing catch block. An ExceptionList is built to describe the failure condition, and the whole message, together with the LocalEnvironment and the newly-populated ExceptionList, is propagated through an exception handling message flow path.

The child of the ExceptionList is always RecoverableException. There is normally only one child of the root, although more than one might be generated in some circumstances. The child of the ExceptionList contains a number of children, the last of which provides further information specific to the type of exception, which can be one of:

- RecoverableException
- ParserException
- ConversionException
- UserException
- DatabaseException

The structure of the exception list tree for a RecoverableException is shown below:



The exception description structure can be both repeated and nested to produce an ExceptionList tree. In this tree:

- The depth (that is, the number of parent-child steps from the root) represents increasingly detailed information for the same exception.
- The width of the tree represents the number of separate exception conditions that occurred before processing was abandoned. This number is usually one, and results in an ExceptionList tree that consists of a number of exception descriptions connected as children of each other.
- At the numbered points in the tree:
 1. This child can be one of RecoverableException, ParserException, DatabaseException, UserException, or ConversionException. All of these have the children shown: if present, the last child is the same element as its parent.
 2. This element might be repeated.
 3. If present, this child contains the same children as its parent.

The children in the tree take the form of a number of name-value elements that give details of the exception, and zero or more name elements whose name is Insert. The NLS (National Language Support) message number identified in a name-value element identifies a WebSphere Message Broker error message. The Insert values are used to replace the variables within this message and provide further detail about the cause of the exception.

The name-value elements within the exception list shown in the figure above are described in the table below.

Name	Type	Description
File ¹	String	C++ source file name
Line ¹	Integer	C++ source file line number
Function ¹	String	C++ source function name
Type ²	String	Source object type
Name ²	String	Source object name
Label ²	String	Source object label
Text ¹	String	Additional text
Catalog ³	String	NLS message catalog name ⁴
Severity ³	Integer	1 = information 2 = warning 3 = error
Number ³	Integer	NLS message number ⁴

Name		Type	Description
Insert ³	Type	Integer	The data type of the value: 0 = Unknown 1 = Boolean 2 = Integer 3 = Float 4 = Decimal 5 = Character 6 = Time 7 = GMT Time 8 = Date 9 = Timestamp 10 = GMT Timestamp 11 = Interval 12 = BLOB 13 = Bit Array 14 = Pointer
	Text	String	The data value
Notes:			
<ol style="list-style-type: none"> 1. Do not use the File, Line, Function, and Text elements for exception handling decision making. These elements ensure that information can be written to a log for use by IBM service personnel. 2. The Type, Name, and Label elements define the object (usually a message flow node) that was processing the message when the exception condition occurred. 3. The Catalog, Severity, and Number elements define an NLS message: the Insert elements that contain the two name-value elements shown define the inserts into that NLS message. 4. NLS message catalog name and NLS message number refer to a translatable message catalog and message number. 			

When the message flow processing is complete, the ExceptionList tree is discarded.

The Airline Reservations sample uses the ExceptionList in the XML_Reservation message flow to pass error information to the Throw node, which generates an error message that includes the information from the ExceptionList.

Correlation names

A correlation name is a field reference that identifies a well-defined starting point in the logical message tree and is used in field references to describe a standard part of the tree format.

When you access data in any of the four trees (message, environment, local environment, or exception list), the correlation names that you can use depend on the node for which you create ESQL or mappings, and whether the node creates an output message. For example, a Trace node does not alter the content of the message as it passes through the node, but a Compute node can construct a new output message.

You can introduce new correlation names with SELECT expressions, quantified predicates, and FOR statements. You can create non-correlation names in a node by using reference variables.

Correlation names in nodes that do not create an output message: Most message flow nodes do not create an output message; all ESQL expressions that you write

in ESQL modules or in mappings within these nodes refer to just the input message. You can use the following correlation names in Database and Filter (ESQL modules) nodes. You cannot use these correlation names in the expression of any mapping for a Mapping, Extract, Warehouse, DataInsert, DataUpdate, or DataDelete node.

Root The root of the message passing through the node.

Body The last child of the root of the message, that is the body of the message. This is an alias for `Root.*[<]`.

For a description of how to use `*` see “Using anonymous field references” on page 178.

DestinationList

The structure that contains the current local environment variables available to the node. Its preferred name is `LocalEnvironment`, although the `DestinationList` correlation name can be used for backward compatibility.

Properties

The standard properties of the input message.

ExceptionList

The structure that contains the current exception list to which the node has access.

Environment

The structure that contains the current global environment variables available to the node. `Environment` can be read and updated from any node for which you can create ESQL code or mappings.

LocalEnvironment

The structure that contains the current local environment variables available to the node. `LocalEnvironment` can be read and updated from any node for which you can create ESQL code or mappings.

Correlation names in nodes that create an output message: If you are coding ESQL for a Compute node, the correlation names are different because there are two message trees involved: the input message and the output message. The correlation names in ESQL within these nodes are:

Environment

The structure that contains the current global environment variables available to the node. `Environment` can be read and updated.

InputRoot

The root of the input message.

InputBody

The last child of the root of the input message. This is an alias for `InputRoot.*[<]`.

For a description of how to use `*` see “Using anonymous field references” on page 178.

InputProperties

The standard properties of the input message.

InputDestinationList

The structure that contains the local environment variables for the message passing through the node. Use the correlation name `InputDestinationList`

for backward compatibility; if compatibility is not required, use the preferred name `InputLocalEnvironment`

InputExceptionList

The structure that contains the exception list for the message passing through the node.

InputLocalEnvironment

The structure that contains the local environment variables for the message passing through the node.

OutputRoot

The root of the output message.

In a Compute node, there is no correlation name `OutputBody`.

OutputDestinationList

The structure that contains the local environment variables being sent out from the node. Use the correlation name `OutputDestinationList` for backward compatibility; if compatibility is not required, use the preferred name `OutputLocalEnvironment`

OutputExceptionList

The structure containing the exception list that the node is generating.

While this correlation name is always valid, it has meaning only when the *Compute Mode* property of the Compute node indicates that the Compute node is propagating the `ExceptionList`.

OutputLocalEnvironment

The structure that contains the local environment variables being sent out from the node.

While this correlation name is always valid, it has meaning only when the *Compute Mode* property of the Compute node indicates that the Compute node is propagating the `LocalEnvironment`.

Predefined and self-defining messages

Each message flowing through your system has a specific structure, which is meaningful to the applications that send or receive that message.

WebSphere Message Broker refers to the structure as the message template. Message template information comprises the *message domain*, *message set*, *message type*, and *physical format* of the message. Together these values identify the structure of the data that the message contains. Every message flow that processes a message conforming to this template must understand the template to enable the message bit stream to be interpreted.

You can use:

- Messages with a message template that you have modeled using the workbench. These are referred to as predefined messages.
- Messages that contain their own template definition. These are called self-defining messages.

Predefined messages: When you create a message using the workbench, you define the fields (Elements) in the message, along with any special field types that you might need, and any specific values (Value Constraints) to which the fields might be restricted.

Every message that you model in the workbench must be a member of a message set. You can group related messages together in a message set: for example, request and response messages for a bank account query can be defined in a single message set.

When you deploy a message set to a broker, the definition of that message set is sent by the Configuration Manager to the broker in the form of a message dictionary. The broker can manage multiple message dictionaries simultaneously.

For information about the benefits of predefining messages, see *Why model messages?*.

The Video Rental sample and the Comma Separated Value (CSV) sample demonstrate how to model messages in XML, CWF, and TDS formats. The EDIFACT sample, FIX sample, SWIFT sample, and X12 sample provide message sets for industry-standard message formats, which might be useful if you use any of those formats.

Self-defining messages: You can create and route messages that are self-defining. These use the XML standard to provide structure to the message, so that it can be interpreted and modified.

Self-defining messages can also be modeled using the workbench. This permits the use of the logical message template by nodes within a message flow. However, you do not have to deploy these message sets to the brokers that support those message flows.

The Large Messaging sample, the Airline Reservations sample, and several other samples in the Samples Gallery use self-defining XML messages for the sake of simplicity; they don't require a message set. The Coordinated Request Reply sample demonstrates how you can transform a message from self-defining XML to a predefined Custom Wire Format (CWF) format, and the Data Warehouse sample demonstrates how you can extract information from an XML message and transform it into BLOB format to store it in a database.

Support for JMS messages: WebSphere Message Broker supports `jms_map` and `jms_stream` messages: it does not support any other category of JMS messages. For further information about using JMS messages with WebSphere Message Broker, see the *WebSphere MQ Using Java* book.

Parsers

A parser is a program that interprets the bit stream of an incoming message and creates an internal representation of the message in a tree structure. It also regenerates a bit stream for an outgoing message from the internal message tree representation.

A parser is invoked when the bit stream that represents an input message is converted to the internal form that can be handled by the broker. The internal form, a logical tree structure, is described in "Logical tree structure" on page 14. The way in which the parser interprets the bit stream is unique to that parser, so the logical message tree that is created from the bit stream varies from parser to parser.

A parser might also create a bit stream from a tree structure if a node in the message flow invokes the `ESQL ASBITSTREAM` function.

The broker requires access to a parser for every message domain to which your input messages and output messages might belong. In addition, it requires a parser for every identifiable message header that might be included in the input or output message. Parsers are invoked as and when required by the message flow.

WebSphere Message Broker provides built-in support for messages in the following message domains by providing the message body parsers listed below:

- MRM (“MRM parser and domain” on page 28)
- XML and XMLNS (“XML parsers and domains” on page 29)
- XMLNSC
- JMSMap and JMSStream (“JMS parser and domains” on page 34)
- IDoc (“IDoc parser and domain” on page 34)
- MIME (“MIME parser and domain” on page 35)
- BLOB (“BLOB parser and domain” on page 46)

It also provides parsers for the following message headers that your applications can include in input messages:

MQCFH

For a list of elements native to this header, see “The MQCFH parser” on page 741

For further information about this header and its contents, see the *WebSphere MQ Programmable Command Formats and Administration Interface* book.

MQCIH

For a list of elements native to this header, see “The MQCIH parser” on page 742

MQDLH

For a list of elements native to this header, see “The MQDLH parser” on page 743

MQIIH

For a list of elements native to this header, see “The MQIIH parser” on page 743

MQMD

For a list of elements native to this header, see “The MQMD parser” on page 744

MQMDE

For a list of elements native to this header, see “The MQMDE parser” on page 745

MQRFH

For a list of elements native to this header, see “The MQRFH parser” on page 745

MQRFH2

For a list of elements native to this header, see “The MQRFH2 parser” on page 746

MQRFH2C

The compact version of the **MQRFH2** parser.

MQRMH

For a list of elements native to this header, see “The MQRMH parser” on page 746

MQSAPH

For a list of elements native to this header, see “The MQSAPH parser” on page 747

MQWIH

For a list of elements native to this header, see “The MQWIH parser” on page 747

SMQ_BMH

For a list of elements native to this header, see “The SMQ_BMH parser” on page 747

If you need to process and parse message body data or headers that the supplied parsers do not handle, create user-defined parsers using the WebSphere Message Broker user-defined parser programming interface.

Warning:

No parser is provided for messages or parts of messages in the format MQFMT_IMS_VAR_STRING. Data in this format is often preceded by an MQIIH header (format MQFMT_IMS). WebSphere Message Broker treats such data as a BLOB. If you change the CodedCharSetId or the Encoding of such a message in a message flow, the MQFMT_IMS_VAR_STRING data is not converted, and the message descriptor or preceding header does not correctly describe that part of the message. If you need the data in these messages to be converted, define the message in the MRM, or provide a user-defined parser.

MRM parser and domain

The MRM message domain includes all messages that are modeled in the workbench.

You can create message models to represent a wide range of message types, with one or more optional physical formats. Messages in this domain are processed by the MRM parser.

The MRM parser is a program that interprets a bit stream or tree that represents a message that belongs to the MRM domain, and generates the corresponding tree from the bit stream on input, or bit stream from the tree on output. Its interpretation depends on the physical format that you have associated with the input or output message:

- For CWF, the parser reads a set sequence of bytes and translates these into the fields and values in the message tree.
- For TDS, the parser uses the Data Element Separation method to parse the bit stream. Depending on the values that you have set for the TDS physical format properties, this might involve identifying delimiters, tags, fixed length elements, patterns, and so on.
- For XML, the parser identifies the XML markup language (tags and attributes) and creates the correct objects, modified by the values that you have set for the XML physical format properties.

In the MRM domain, the message is considered in two parts:

1. The logical message model. This is the piece of the message that conveys the business data, devoid of its physical representation (how it appears in a bit stream on the wire). It is independent of platform and the way in which the message is constructed.

For example, if you define a message that conveys information about a debit of an individual's bank account, it can be represented in different physical forms on the wire (in XML, or a fixed structure such as a COBOL copybook). The business meaning and data is the same in both cases: only the physical layout has changed.

2. The physical representation. This is how the data is physically laid out on the wire. A single logical message model might have several different ways in which it can be physically represented.

This two-part definition can be useful, because it handles situations in which you need to connect two different systems. For example, a legacy style application that expects data to be passed to it in the form of COBOL copybooks, might need to communicate with a system that expects data in the form of XML. Both applications work with the same data, and it would be undesirable to alter either application. By routing the messages through a broker, you can use a single logical model with multiple physical representations to provide the required transformation.

The Video Rental sample, Comma Separated Value (CSV) sample, EDIFACT sample, FIX sample, SWIFT sample, and X12 sample all use the MRM parser to process messages.

XML parsers and domains

The XML message domain includes all messages that conform to the W3C XML standard.

Messages in this domain are processed by one of the XML, XMLNS or XMLNSC parser. The XMLNS domain is an extension of the XML domain and contains messages that conform to the same standard and that exploit the namespaces feature of the XML specification. Messages in this domain are processed by the XML parser.

The XML parser is a program that interprets a bit stream or tree that represents a message that belongs to the XML domain and generates the corresponding tree from the bit stream on input, or bit stream from the tree on output. The bit stream is a representation of an XML file. (The XML parser also interprets a bit stream or tree that represents a message that belongs to the JMS domains; there is no JMS parser.)

Your applications can exchange XML messages (with or without namespace support) with the WebSphere Message Broker brokers in two ways:

1. You can predefine (model) the message template to create a message dictionary. If you do so, your XML messages are parsed by the MRM parser and processed in the same way as all messages that you model.
2. You can use self-defining messages that you do not specify in any way before sending.

A self-defining message can be handled by every built-in node. The whole message can be stored in a database, and headers can be added to or removed from the message as it passes through the message flow.

A self-defining message is also known as a generic XML message. It does not have a recorded format, but carries the information about its content and structure within the message in the form of a document that adheres to the XML specification. Its definition is not held anywhere else. When an XML message is received by the broker, it is interpreted by the XML parser, and an internal message tree structure is created according to the XML definitions contained within that message.

Details of how the XML parser handles null elements and values is described in "The XML parser and null values" on page 33.

The information provided with WebSphere Message Broker does not provide a full definition or description of XML terminology, concepts, and message constructs: it is a summary that highlights aspects that are important when you use XML messages with brokers and message flows.

For further information about XML, see the developerWorks Web site.

Example XML message parsing: The name elements used in this description (for example, XmlDecl) are provided by WebSphere Message Broker and are referred to as correlation names. They are available for symbolic use within the ESQL that defines the processing of message content performed by the nodes, such as a Compute or Filter node, within a message flow. They are not part of the XML specification. Each XML parser defines its own set of correlation names because the handling of XML content varies.

The correlation names for XML name elements (for example, Element and XmlDecl) equate to a constant value of the form 0x01000000 etc. You can see these constants used in the output created by the Trace node when a message, or a portion of the message, is traced.

A simple XML message might take the form:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Envelope
PUBLIC "http://www.ibm.com/dtds" "example.dtd"
[<!ENTITY Example_ID "ST_TimeoutNodes Timeout Request Input Test Message">]
>
<Envelope version="1.0">
  <Header>
    <Example>&Example_ID;</Example>
    <!-- This is a comment -->
  </Header>
  <Body version="1.0">
    <Element01>Value01</Element01>
    <Element02/>
    <Element03>
      <Repeated>ValueA</Repeated>
      <Repeated>ValueB</Repeated>
    </Element03>
    <Element04><P>This is <B>bold</B> text</P></Element04>
  </Body>
</Envelope>
```

The following sections show the output created by the Trace node when the above message has been parsed in the XML and XMLNSC parsers to demonstrate the differences in the internal structures used to represent the data as it is processed by the Broker.

Example XML Message parsed in the XML domain: Note in the following the WhiteSpace elements within the tree are there because of the space, tab and line breaks that format the original XML document, for presentation clarity the actual characters in the trace have been replaced with 'WhiteSpace'. WhiteSpace within an XML element does have business meaning and is represented using the Content syntax element. Note that the XmlDecl, DTD, and comments are represented in the XML domain using explicit correlation named syntax elements.

```
(0x01000010):XML = (
  (0x05000018):XML = (
    (0x06000011): = '1.0'
    (0x06000012): = 'UTF-8'
    (0x06000014): = 'no'
  )
  (0x06000002): = 'WhiteSpace'
  (0x05000020):Envelope = (
    (0x06000004): = 'http://www.ibm.com/dtds'
    (0x06000008): = 'example.dtd'
    (0x05000021): = (
      (0x05000011):Example_ID = (
        (0x06000041): = 'ST_TimeoutNodes Timeout Request Input Test Message'
      )
    )
  )
  (0x06000002): = 'WhiteSpace'
  (0x01000000):Envelope = (
    (0x03000000):version = '1.0'
    (0x02000000): = 'WhiteSpace'
    (0x01000000):Header = (
      (0x02000000): = 'WhiteSpace'
      (0x01000000):Example = (
        (0x06000020): = 'Example_ID'
        (0x02000000): = 'ST_TimeoutNodes Timeout Request Input Test Message'
        (0x06000021): = 'Example_ID'
      )
      (0x02000000): = 'WhiteSpace'
      (0x06000018): = ' This is a comment '
      (0x02000000): = 'WhiteSpace'
    )
    (0x02000000): = 'WhiteSpace'
    (0x01000000):Body = (
      (0x03000000):version = '1.0'
      (0x02000000): = 'WhiteSpace'
      (0x01000000):Element01 = (
        (0x02000000): = 'Value01'
      )
      (0x02000000): = 'WhiteSpace'
      (0x01000000):Element02 =
      (0x02000000): = 'WhiteSpace'
      (0x01000000):Element03 = (
        (0x02000000): = 'WhiteSpace'
        (0x01000000):Repeated = (
          (0x02000000): = 'ValueA'
        )
      )
      (0x02000000): = 'WhiteSpace'
      (0x01000000):Repeated = (
        (0x02000000): = 'ValueB'
      )
    )
    (0x02000000): = 'WhiteSpace'
  )
  (0x02000000): = 'WhiteSpace'
  (0x01000000):Element04 = (
    (0x01000000):P = (
      (0x02000000): = 'This is '
      (0x01000000):B = (
        (0x02000000): = 'bold'
      )
    )
  )
)
```

```

    )
    (0x02000000): = ' text'
  )
)
(0x02000000):      = 'WhiteSpace'
)
(0x02000000):      = 'WhiteSpace'
)

```

Example XML Message parsed in the XMLNSC domain: The following trace shows the elements created to represent the same XML structure within the Compact XMLNSC parser in its default mode. In this mode the compact parser does not retain comments, processing instructions or mixed text.

It can be clearly seen by comparison that there is a large saving in the number of syntax elements used to represent the same business content of the Example XML message by using the compact parser.

Note that by not retaining mixed text all of the WhiteSpace elements that have no business data content are no longer taking any runtime foot print in the Broker message tree. However this also results in the mixed text in "Element04.P" being discarded and only the value of the child folder "Element04.P.B" is held in the tree, the text 'This is ' and ' text' in "P" is discarded . This type of XML structure is not normally associated with business data formats so use of the compact XMLNSC parser will generally be desirable. However should you need this type of processing you would either not use the XMLNSC parser or use it with the "retain mixed text" mode enabled.

The handling of the XML declaration is also different in the compact parser with the version, encoding and standalone attributes being held as children of the XmlDeclaration rather than special correlation named elements.

```

(0x01000000):XMLNSC = (
  (0x01000400):XmlDeclaration = (
    (0x03000100):Version = '1.0'
    (0x03000100):Encoding = 'UTF-8'
    (0x03000100):StandAlone = 'no'
  )
  (0x01000000):Envelope = (
    (0x03000100):version = '1.0'
    (0x01000000):Header = (
      (0x03000000):Example = 'ST_TimeoutNodes Timeout Request Input Test Message'
    )
    (0x01000000):Body = (
      (0x03000100):version = '1.0'
      (0x03000000):Element01 = 'Value01'
      (0x01000000):Element02 =
      (0x01000000):Element03 = (
        (0x03000000):Repeated = 'ValueA'
        (0x03000000):Repeated = 'ValueB'
      )
      (0x01000000):Element04 = (
        (0x01000000):P = (
          (0x03000000):B = 'bold'
        )
      )
    )
  )
)

```

Most of the samples in the Samples Gallery use the XML parser to process messages. For example, have a look at the Coordinated Request Reply sample, Large Messaging sample, and Message Routing sample.

The XML parser and null values:

This topic describes how the XML parser handles explicit nulls.

The XML domain does not have a concept of a null value in the message bit stream. There is no sequence of bytes in an XML message that leads to an explicit null value being created in the message tree by the XML parser. Although the parser does not create a message tree field with an explicit null value, you can code ESQL within a message flow that creates a field of this form.

When the XML parser constructs a new message bit stream from the message tree, it can encounter explicit null values and must be able to handle them. In this situation, an explicit null value is seen as "no characters to write" and the field is created as the empty tags.

In the XML domain, there are several values that lead to writing XML empty tags in the output XML message:

- A message tree field that contains an explicit null value.
- A message tree field that contains the zero length string. (' ')
- A message tree field that does not have a value.

When these have been written as the empty tags in an XML message, if the XML message is subsequently parsed, it is not possible to distinguish between these three cases. When empty tags are parsed in XML, an XML message tree field is created that does not have a value. This is different from a message tree field that contains an explicit null value, and one that contains a zero length string.

Although the XML parser resolves these three different message tree field values to the same result in the message bit stream, other message tree operations can distinguish between them. Although the XML parser does not create fields that contain explicit nulls and zero length strings, you can perform operations in message flows that can.

Therefore, if a message flow is processing messages in the XML domain, the ESQL or Java must distinguish between an explicit null value, a zero length string, and a message tree field that does not have a value. Not even a field that contains the explicit null value is considered to be null when you code ESQL to query the value of the element.

The XML writing process can handle this condition, but other operations within ESQL do not treat this as a null value. The XML domain does not have a concept of null and the parser never creates a field that contains an explicit null value. Because the XML domain is a character-based domain, when the value of a field is queried, it returns the result as a character field. If you create an XML message tree field with an explicit null value, this is not seen as a null value for those ESQL routines that get the value of the message tree field.

What is returned is the character representation of the contents of the field, which in this case is the character string 'NULL'. For example, if you perform such comparisons with IS NULL, this never evaluates to a TRUE value. If you copy this message tree field to another message tree, the target message tree field is populated with the character string 'NULL'.

JMS parser and domains

The JMS message domains include all messages that are produced by the WebSphere MQ implementation of the Java Message Service (JMS) API standard.

These messages, which have a message type of either JMSMap or JMSStream, are supported in the same way as messages in the XML domain and are parsed by the XML parser (there is no JMS parser).

Your applications can exchange JMS messages with the broker in two ways:

1. You can predefine the message template using MRM facilities to create a message dictionary. If you do so, your JMS messages are handled in the same way as all messages that you predefine to the MRM, and you can take advantage of message validation and other manipulation within the message flow that is available when the message definition is known.
2. You can use self-defining messages that you do not specify in any way before sending.

A self-defining message can be handled by every built-in node. The whole message can be stored in a database, and headers can be added to or removed from the message as it passes through the message flow.

When you include an input node, you can specify the domain to which the message belongs as a property of the node. You can choose from the values JMSMap and JMSStream. Messages in these domains are passed to the XML parser by the input node.

IDoc parser and domain

The IDoc domain processes messages that are sent to the broker by SAP R3 clients across the MQSeries[®] link for R3. It also allows data of another format supported by the broker (for example, JMS) to be mapped to an IDoc stream.

The class name of the parser that is needed is defined in the Format field in the MQMD header of the input message. The parser creates the message tree from an input message in a specified format. This format must be followed when a message tree is built in a WebSphere Message Broker message flow.

For a SAP IDoc message the name of the high level element (the last child of the root element) must be IDOC to match the message domain supported by the parser.

A typical IDoc message structure that has been sent from SAP to the MQSeries link for R3 is represented in WebSphere Message Broker as the root element followed by:

- A message descriptor (MQMD), which is given to the MQMD parser.
- An MQSAPH header, which is given to the MQSAPH parser.
- The IDoc parser, which is made up of fixed size structures:
 - The first structure is the Control Structure (DC) which is 524 bytes long and contains name-value pairs for the elements of the DC.
 - One or more Data Structures (DDs). Each data structure is 1063 bytes long and is composed of name-value pairs, with the last field of the structure, which is 1000 bytes long, holding the segment data.

Use the MRM to model each individual segment.

MIME parser and domain

The MIME parser does not support the full MIME standard but does support common uses of MIME, including SOAP with Attachments (SwA). The messages can be sent to the broker over HTTP or over other transport types such as WebSphere MQ. Use the MIME domain if your messages use the MIME standard for multipart messages.

The multipurpose internet mail extensions (MIME) domain does not support Content-Type values with a media type of message.

To specify that a message uses the MIME domain use one of the following methods:

- Select a Message Domain of MIME on the relevant message flow nodes:
 - input nodes
 - HTTPRequest nodes
 - MQGET nodes
 - ResetContentDescriptor nodes
- If you are using WebSphere MQ, supply the domain MIME in the MQRFH2 header.

The MIME domain and parser allow you to parse and write MIME messages. The MIME parser creates a logical tree and sets up the broker ContentType property. You can use Compute nodes, Java Compute nodes, and Mapping nodes to manipulate the logical tree. Set the Content-Type value using the ContentType property in the MIME domain.

Example MIME message

The example below shows a simple multipart MIME message. The message shown is a SOAP with attachments message with two parts, the root part and one attachment part. The boundary string *MIME_boundary* delimits the parts.

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml
Content-Description: Optional description of message.
```

```
Optional preamble text
--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <rootpart@example.com>
```

```
<?xml version='1.0' ?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

  <SOAP-ENV:Header xmlns:ins="http://myInsurers.com">
    <ins:ClaimReference>abc-123</ins:ClaimReference>
  </SOAP-ENV:Header>

  <SOAP-ENV:Body xmlns:ins="http://myInsurers.com">
    <ins:SendClaim>
      <ins:ClaimDetail>myClaimDetails</ins:ClaimDetail>
      <ins:ClaimPhoto>
        <href>cid:claimphoto@example.com</href>
      </ins:ClaimPhoto>
    </ins:SendClaim>
  </SOAP-ENV:Body>

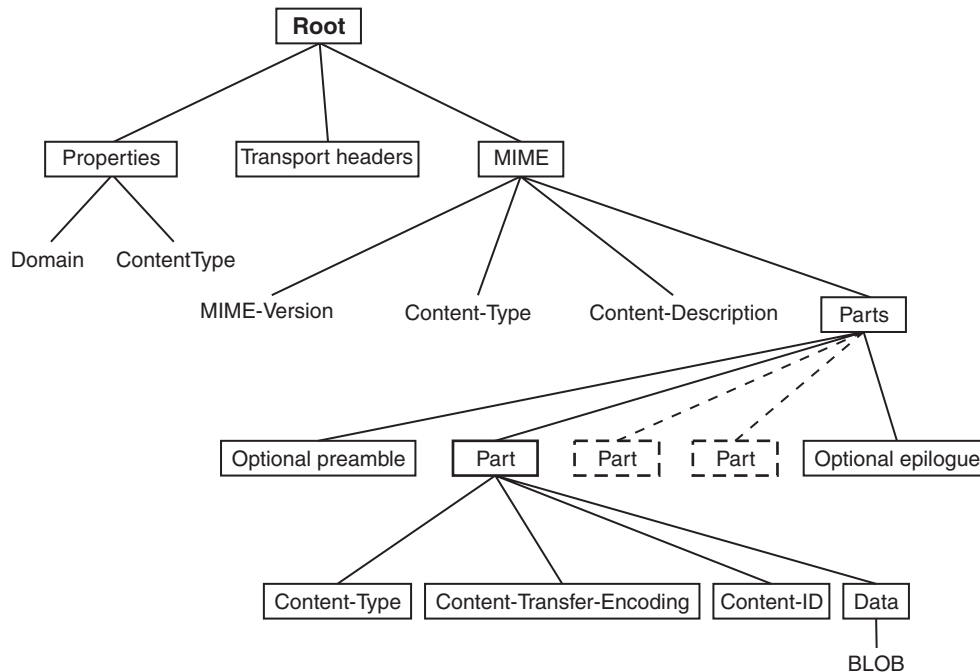
</SOAP-ENV:Envelope>
```

```
--MIME_boundary
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary
Content-ID: <claimphoto@example.com>
```

```
myBinaryData
--MIME_boundary--
Optional epilogue text
```

Example MIME logical tree

The diagram below is an example of a MIME logical tree. A MIME logical tree does not need to contain all of the children shown here. The value of the Content-Type header of a MIME message is the same as the ContentType field in the Properties subtree. The Transport-headers are headers from the transport used, such as an MQMD.



You can parse the BLOB data in the tree further if you know about the format of that MIME part. You might be able to find information about the format from its Content-Type field in the logical tree. Alternatively, you might know the format that your MIME messages take and be able to parse them appropriately. For example, you might know that the first MIME Part is always an XML message and that the second MIME Part is a binary security signature.

You must specify how to parse other message formats, such as tagged delimited or binary data, within your message flow, because the MIME parser does not do this. You must also specify how to handle encoded and signed message parts, because the MIME parser does not process these.

**MIME messages:
MIME headers**

A MIME message consists of both data and metadata. MIME metadata consists of HTTP-style headers and MIME boundary delimiters. Each header is a colon-separated name-value pair on a line. The ASCII sequence <CR><LF> terminates the line. A sequence of these headers, called a header block, is terminated by a blank line: <CR><LF><CR><LF>. Any headers following this HTTP-style can appear in a MIME document. A number of standard MIME headers are described in MIME standard header fields.

Content-Type

The only header that the MIME parser insists on being present is the *Content-Type* header. This header specifies the type of the data in the message. If the Content-Type value starts with “multipart” then the message is a multipart MIME message. For multipart messages the Content-Type must also include a boundary attribute giving the text used to delimit the message parts. Each separate MIME Part has its own Content-Type field that specifies the type of the data in the Part.

This may also be multipart, allowing multipart messages to be nested. MIME parts with any other Content-Type values are handled as BLOB data.

If a MIME document is sent over HTTP, then the Content-Type header appears in the HTTP header block rather than in the MIME message body. For this reason, the broker manages the value of the Content-Type header as the ContentType property in the *Properties* folder of the logical tree. This allows the MIME parser to obtain the value of Content-Type for a MIME document received over HTTP. If you need to either create a new MIME tree or modify the value of the Content-Type. Set the Content-Type value using the ContentType property in the MIME domain. Setting the Content-Type value directly in the MIME tree or HTTP trees can lead to the value being ignored or used inconsistently. The following ESQL is an example of how to set the broker ContentType property:

```
SET OutputRoot.Properties.ContentType = 'text/plain';
```

Parsing

The MIME domain does not enforce the full MIME specification. This allows you to work with messages that might be invalid in other applications. For instance, the MIME parser does not insist on a **MIME-Version** header. The MIME parser imposes these constraints:

- The MIME headers must be properly formatted. This means that:
 - Each header is a colon-separated name-value pair, on a line of its own, terminated by the ASCII sequence <CR><LF>.
 - The header line must use 7-bit ASCII.
 - Use semicolons to separate parameters:
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml
 - A header may have a comment in parentheses, for example:
MIME-Version: 1.0 (Generated by XYZ)
- A line starting with white space is treated as a continuation of the previous line. This allows a long header to be split across multiple lines.
- If two or more headers in a header block have the same name then their values are concatenated into a comma separated list.
- A top level MIME Content-Type header must be available. The header is not case-sensitive. If the transport is HTTP, any Content-Type value in the HTTP header is used as the top-level Content-Type. If the transport is not HTTP, the Content-Type must appear in the initial header block of the MIME message.
- The Content-Type value is a media type followed by the / character and a subtype. Examples of this are text/xml and multipart/related. The parser does not validate subtypes. This value can be followed by one or more semicolon separated parameters.
- If the media type of a message is multipart then a boundary attribute must provide the text used to delimit the separate MIME parts.
- Each individual MIME part can have its own Content-Type header. The Part header can have a media type of multipart, allowing multipart messages to be nested. In this case a valid boundary attribute must be provided and its value must be different from any previously defined in the message. MIME parts with any other Content-Type value are handled as BLOB data.
- MIME multipart boundary delimiters are represented in 7-bit ASCII. The boundary delimiter consists of a line starting with a hyphen pair, followed by a boundary string. This sequence must not occur within the MIME message at any point other than as a boundary. A MIME end-delimiter is a hyphen pair,

followed by the MIME boundary string, followed by a further hyphen pair. All delimiter lines must end in the ASCII sequence <CR><LF>. An example of a delimited message is:

```
--MIME_boundary
message_data
--MIME_boundary
message_data
--MIME_boundary--
```

where *MIME_boundary* is the boundary delimiter string, and *message_data* represents message data.

- The MIME media type “*message*” is not supported and results in an error at run time.
- Any preamble data (text between the initial MIME header block and the first boundary delimiter) or epilogue data (text after the final boundary delimiter) is stored in the logical tree as a value-only element. Preamble data and epilogue data can only appear as the first and last children of a Parts node respectively.
- The MIME parser does not support on demand parsing and ignores the Parse Timing property. The parser does not validate MIME messages against a message model and ignores the Message Brokers Toolkit validate property.

Special cases of multipart MIME

The MIME parser is intended primarily for use with multipart MIME messages. However, the parser also handles some special cases:

- Multipart MIME with just one part. The logical tree for the MIME part saves the Content-Type and other information as usual, but the Data element for the attachment is empty.
- Single Part MIME. For single part MIME the logical tree has no Parts child. The last child of the MIME tree is the Data element. The Data element is the parent of the BLOB that contains the message data.
- MIME Parts with no content.

Secure MIME (S/MIME)

S/MIME is a standard for sending secure e-mail. S/MIME has an outer level Content-Type of *multipart/signed* with parameters **protocol** and **micalg** that define the algorithms used to encrypt the message. One or more MIME parts can have encoded content. These parts have Content-Type values such as *application/pkcs7-signature* and a Content-Transfer-Encoding of *base64*. The MIME domain does not attempt to interpret or verify whether the message is actually signed.

MIME tree details:

Logical tree elements

A MIME message is represented in the broker as a logical tree with the following elements:

- The root of the tree is a node called MIME.
- All correctly formatted headers are stored in the logical tree, regardless of whether they conform to the MIME standard. The headers appear in the logical tree as name=value, as shown here:

```
Content-Type=text/xml
```

- A multipart MIME message is represented by a subtree with a root node called Parts.
- Any preamble or epilogue data associated with a multipart MIME message is represented by value-only elements appearing as the first and last children of Parts.
- In the special case of single-part MIME, the content is represented by a subtree with the root called Data.
- Each part of a multipart MIME message is represented by an element called Part with a child element for each MIME header, and a last child called Data.
- The Data element represents the content of a MIME part. This makes it easier to test for the presence of body content using ESQL because the Data element is always the last child of its parent.

Writing MIME messages

When writing a message, the MIME parser creates a message bit stream using the logical message tree. The MIME domain does not enforce all of the constraints that the MIME specification requires, therefore it might generate MIME messages that do not conform to the MIME specification. The constraints that the MIME parser imposes are:

- The tree must have a root called MIME, and constituent Parts, Part, and Data elements, as described in “Logical tree elements” on page 39.
- Exactly one Content-Type header must be present at the top level of the tree, or be available via the ContentType property. Media subtypes are not validated.
- If the media type is *multipart* then there must also be a valid boundary parameter.
- Any constituent MIME parts may have exactly one Content-Type header. If the value of this header starts with *multipart* then it must also include a valid boundary parameter. The value of this boundary parameter must not be the same as other boundary parameter values in the definition.
- The MIME Content-Type value “message” is not supported and results in an error at run time.
- All name=value elements in the tree are written as name: value followed by the ASCII sequence <CR><LF>.

If you have other elements in the tree, the parser behaves in the same way as the HTTP header parser:

- A name-only element or a NameValue element with a NULL value results in Name: NULL .
- Any children of a name=value element are ignored.

The message flow must serialize subtrees if they exist. This can be done using the ESQL command ASBITSTREAM.

Developing flows using the MIME domain:

The following topics describe scenarios showing how MIME messages might be used in a message flow. The scenarios concentrate on the details that are specific to MIME.

In all these scenarios an external application might enforce the MIME standard more strictly than the broker MIME parser. For example an application might insist on the presence of a MIME-Version header.

The scenarios are:

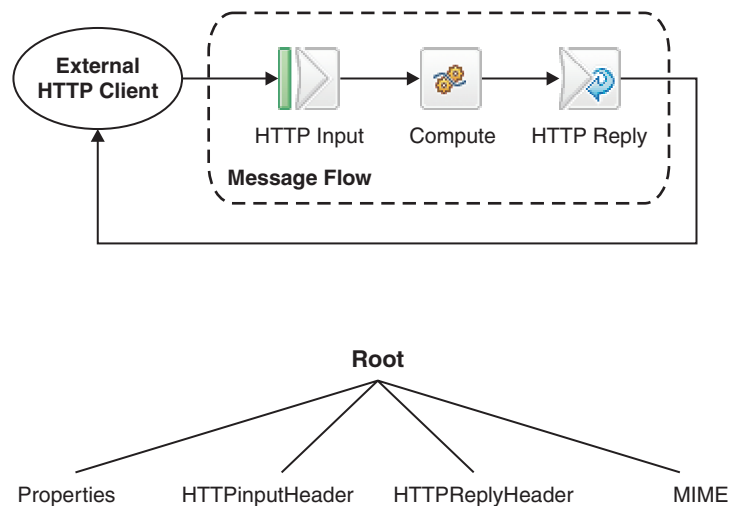
- “Creating a Web service with HTTP as the transport”
- “Creating a Web service with WebSphere MQ as the transport” on page 42
- “Accessing a WebSphere MQ enabled application as a Web service” on page 42
- “A WebSphere MQ enabled application calling a Web service” on page 44

MIME messages can be sent and received over various transports, including HTTP and WebSphere MQ. The following samples illustrate transport specific details. For example, creating headers and saving and restoring message correlators, such as the HTTP request identifier:

- Web Service Host sample
- Coordinated Request Reply sample

Creating a Web service with HTTP as the transport:

This scenario implements a Web service using HTTP as the transport mechanism and MIME as the domain. A message flow for this scenario, and the resulting message tree are given below:



When a MIME message enters the message flow the top-level Content-Type of the message is stored in the HTTPInputHeader tree and in the MIME tree. The broker also stores a copy of the Content-Type of the message as the ContentType value in the Properties subtree.

Any processing that this message flow needs to do is done in the Compute node. The output domain of this message flow is also MIME, therefore the output message must be a MIME tree. This tree can be made by either creating a new tree, or modifying the incoming MIME message tree using the Compute node. If the Content-Type of a message needs to be modified, update the broker ContentType property. When this property is changed, the MIME tree is updated automatically.

For the message to be output as an HTTP reply, there must be a HTTP reply header. You can create this in two ways:

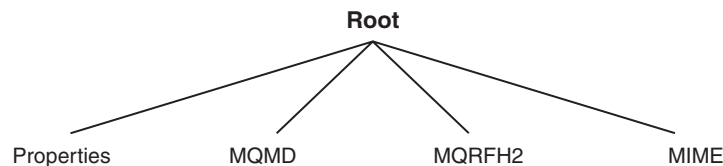
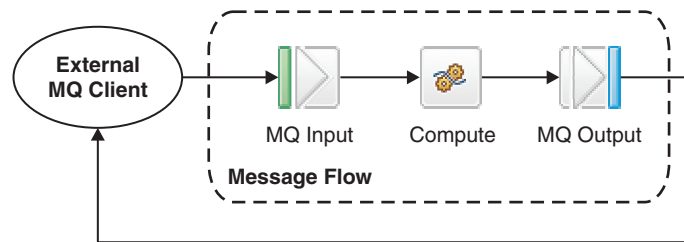
- Set the HTTPReply node to automatically generate the header:

1. Right-click the HTTPReply node and click **Properties**.
 2. Click the **Generate default HTTP headers from input or response** option in the Basic properties.
- Set up an HTTPReplyHeader in the output tree, as illustrated in the following ESQL:


```
SET OutputRoot.HTTPReplyHeader.Host = 'localhost:1234';
```

Creating a Web service with WebSphere MQ as the transport:

This scenario implements a Web service using WebSphere MQ as the transport mechanism and MIME as the domain. A message flow for this scenario, and the resulting message tree are given below:



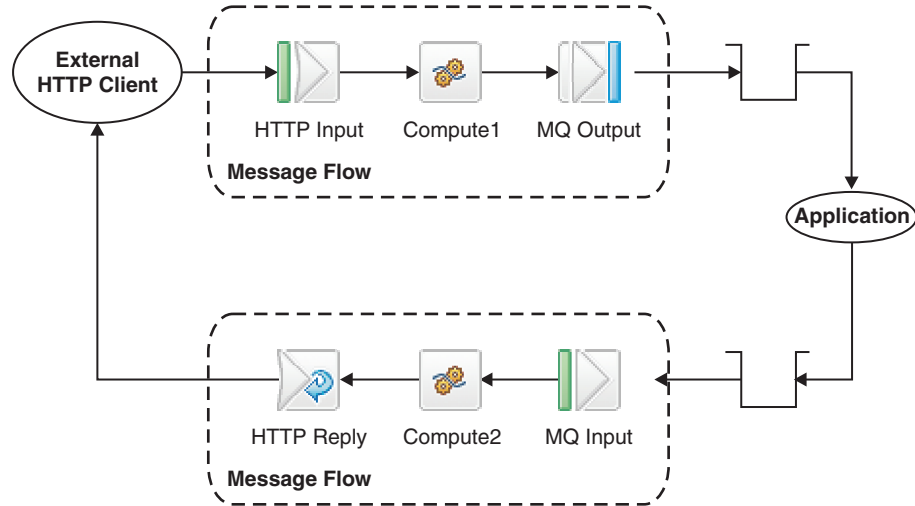
When a MIME message enters the message flow the Content-Type and any other top-level MIME headers are stored in the MIME tree. The broker also stores a copy of the Content-Type of the message as the ContentType value in the Properties subtree.

Any processing that this message flow needs to do is done in the Compute node. The output domain of this message flow is also MIME, therefore the output message must be a MIME tree. This tree can be made by either creating a new tree, or modifying the incoming MIME message tree using the Compute node. If the Content-Type of a message needs to be modified, updating the broker ContentType property. When this property is changed, the MIME tree is updated automatically.

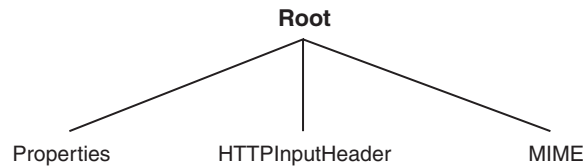
The Coordinated Request Reply sample sample contains information about manipulating MQ headers and using the MQGet node.

Accessing a WebSphere MQ enabled application as a Web service:

This scenario represents a Web service providing an interface to a WebSphere MQ enabled application using MIME as the domain. A message flow for this scenario, and the resulting message trees are given below:



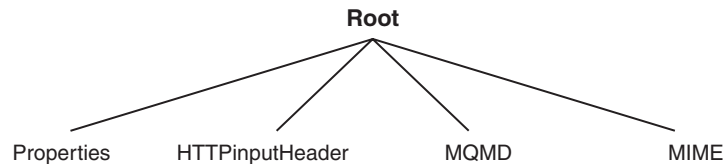
When a MIME message enters the message flow the top-level Content-Type of the message is stored in the HTTPInputHeader tree and in the MIME logical message tree. The broker also stores a copy of the Content-Type of the message as the ContentType value in the Properties subtree. The diagram below shows the message tree after the message has left the HTTPInput node:



The first Compute node, Compute1, must add an MQMD header to the message and save the HTTP correlator for the return flow to use. The HTTP correlator could be stored in a database, or copied to the message body. The following example ESQL illustrates how the correlator can be stored in an XML message body:

```
SET OutputRoot.XML.X.rid =
  CAST(InputLocalEnvironment.Destination.HTTP.RequestIdentifier AS CHARACTER);
```

The diagram below shows the message tree after it has left Compute1:



If the application that is receiving the MQ message expects the message to be in a MIME format, a MIME tree is needed. Compute1 can supply this by either creating

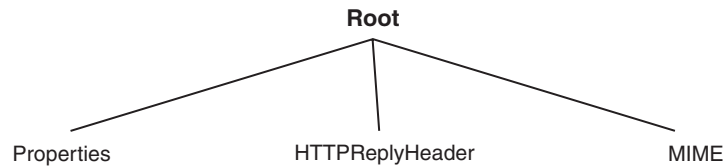
a new MIME tree, or modifying and propagating the existing MIME tree. To modify the Content-Type of the message, the broker ContentType property should be used. When the broker ContentType property is modified, the Content-Type property in the MIME tree is updated automatically.

When the message is received from the message queue via the MQInput node, the second Compute node, Compute2, must remove the MQMD header from the message and restore the HTTP correlator. If Compute1 copied the correlator from the message body as described above, Compute2 can restore the correlator with the following ESQL:

```
SET OutputLocalEnvironment.Destination.HTTP.RequestIdentifier = CAST(InputRoot.XML.X.rid AS BLOB);
```

Compute2 can also set up an explicit HTTPReplyHeader.

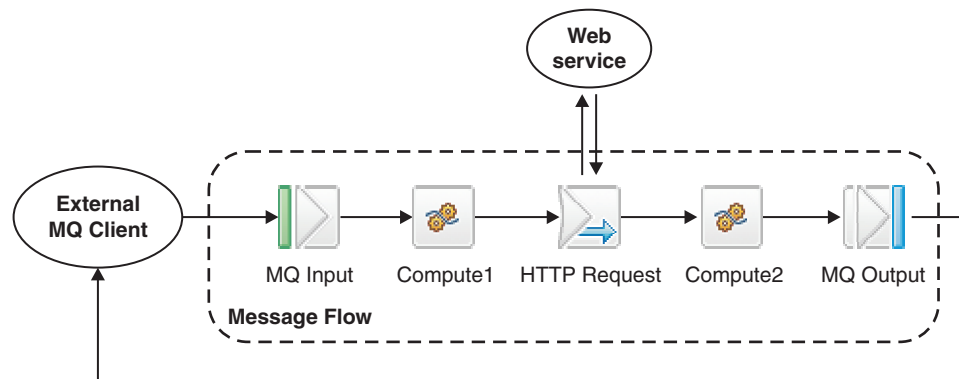
If the output domain is MIME, a MIME tree must be created to output the message. Compute2 either creates a new MIME message, or modifies and propagates the input MIME message as required to create the output message. The following diagram shows the message tree after it has gone through Compute2:



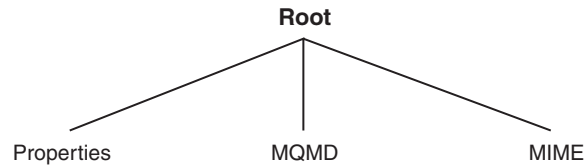
Many variations of this scenario are possible. For instance, a single flow could be created using an MQGet node instead of the MQInput node and then the HTTP correlator would not need to be saved. However, this is potentially less scalable. The Coordinated Request Reply sample gives further information on using the MQGet node.

A WebSphere MQ enabled application calling a Web service:

This scenario represents an application that uses WebSphere MQ to make a call to a Web service while processing a message. The MIME domain is used for this example. A message flow for this scenario, and the resulting message trees are given below:



When a MIME message enters the message flow the Content-Type and any other top-level MIME headers are stored in the MIME tree. The broker also stores a copy of the Content-Type of the message as the ContentType value in the Properties subtree. The diagram below shows some of the message tree after it has left the MQInput node:



The first Compute node, Compute1, is used to set up the HTTPRequestHeader if one is required. Compute 1 can also be used to create a new MIME tree or to modify the existing MIME tree if the intermediate application providing the Web service requires a MIME message.

When the HTTPRequest node makes a request message, it removes the MQMD header from the message tree. If you need to save the information from the MQMD, such as the *MsgId*, to use in the reply message to the MQ client, you can do it in one of the following ways:

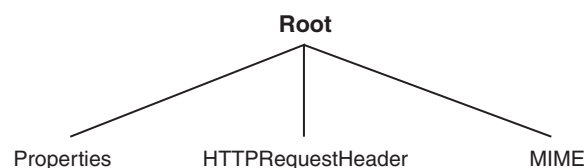
- Compute1 can save the required fields in the Environment tree so that the second Compute node, Compute2, can re-create the MQMD.
- Compute1 saves the required fields into the OutputLocalEnvironment tree so that Compute2 can re-create the MQMD. To do this, Compute1 must be configured to propagate both Message and LocalEnvironment.
- Configure the HTTPRequest node to not replace the input message with the HTTP response. Instead specify that the response should be attached as OutputRoot.MIME. The original input tree and MQMD are then still available, but Compute2 needs to alter the tree before passing the message to the MQOutput node. For example, Compute2 might need to remove parts of the tree, such as HTTPRequestHeader and HTTPResponseHeader. To do this, copy just the parts of the tree that you do want to keep. The following ESQL shows an example of how to do this:

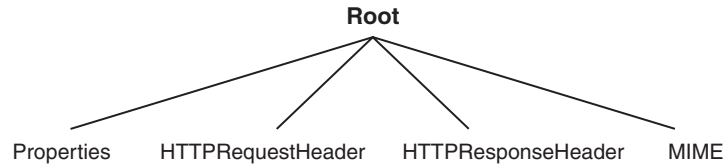
```

SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.MIME = InputRoot.MIME;
  
```

HTTPRequest node properties control the content of output tree.

The diagrams below show the message tree before and after making the Web service call in the HTTPRequest node:





Compute2 is used to create or restore an MQMD if one is needed, and to tidy up the tree to remove inappropriate headers such as the HTTPResponseHeader.

BLOB parser and domain

The BLOB message domain includes all messages with content that cannot be interpreted and subdivided into smaller sections of information.

Messages in this domain are processed by the BLOB parser. The BLOB parser is a program that interprets a bit stream or message tree that represents a message that belongs to the BLOB domain, and generates the corresponding tree from the bit stream on input, or a bit stream from the tree on output.

A BLOB message is handled as a single byte string, and although you can manipulate it, you cannot identify specific pieces of the byte string using any reference as you can with messages in other domains.

You can process messages in the BLOB domain in the following ways:

- You can refer to the message content if you know the location (offset) of particular information within the message. You can specify offset values in ESQL statements within nodes in a message flow to manipulate the information.
- You can store the message in an external database, in whole or in part (where the part is identified by the offset of the data to be stored).

The BLOB message body parser does not create a tree structure in the same way that other message body parsers do. It has a root element BLOB, that has a child element, also called BLOB, that contains the data.

For example, `InputBody.BLOB.BLOB[10]` identifies the tenth byte of the message body; `substring(InputBody.BLOB.BLOB from 10 for 10)` references 10 bytes of the message data starting at offset 10.

The Data Warehouse sample demonstrates how you can extract information from an XML message and transform it into BLOB format to store it in a database.

Null handling

An input message can contain null fields and values, and a message flow can create a logical tree for an output message that contains null fields and values.

Within a logical tree, there are two types of null:

Implicit

A field does not exist and has a null value by implication.

Explicit

A field contains a specific value that is interpreted as null.

The way in which an explicit null value is interpreted depends on the parser, the operations performed by the parser, and the ESQL operations performed on the message tree field.

You can use the ESQL SET and CREATE statements to populate a message tree field with a name, type, and value. The value assigned to the message tree might therefore be the result of an expression that evaluates to a null. This can be caused by the use of an uninitialized variable, a message tree field that does not exist, the result of a database query, or the explicit use of the NULL keyword.

Explicit nulls are handled by the MRM and generic XML parsers:

- The parser identifies a sequence of bytes in the input message bit stream and inserts a null value in the related message tree field.
- The parser accepts an explicit null value in the message tree and generates a sequence of bytes in the output message bit stream.

The following considerations for explicit nulls apply:

- Null is not a consistent concept. It has different values in different contexts.
- The concept of null is present in the logical tree and its value is represented in the logical message tree where appropriate; there is no concept of a null value in a bit stream. The null value in the message tree is created when a specific sequence of bytes is encountered within the bit stream. This sequence of bytes is different for each parser.
- Each parser has its own representation of a null in a bit stream. In the MRM domain, you can specify a different representation for each element if you choose.
- Each parser interprets a null value in a message tree field differently.

In summary, some parsers create a null value in the message tree from the input bit stream, and can handle a message tree field containing an explicit null value when they write a new output message bit stream. However, you can perform other operations on the fields in the message tree during message processing, and you can configure a message flow to copy message tree fields from one domain to another.

The following topics provide more information about NULL handling:

- “The XML parser and null values” on page 33
- Custom wire format: NULL handling
- XML wire format: NULL handling
- TDS format: NULL handling

Which parser should you use?

The characteristics of the messages that your applications exchange indicate which parser you must use.

WebSphere Message Broker provides a range of message parsers. Each parser processes message body data for messages in a particular message domain (for example, XML), or a particular message header (for example, the MQMD).

Review the messages that your applications send to the broker, and determine which message domain the message body data belongs to, so that you can either set up the correct headers in the message, or configure the input node of the message flow appropriately.

If your application data is in XML format

Use either the XML format in the MRM domain, or the XML, XMLNS or XMLNSC domain.

Usually, you will find that the MRM domain offers greater facilities:

- When a message is parsed, the logical message tree uses the types taken from the message model. This allows ESQL expressions to operate on the data directly, without having to cast it to the correct data type. Data encoded in CData sections is supported, as well as binary data in hexadecimal and base64 encoding.
- When you create ESQL to configure a Compute, Database, or Filter node, the ESQL editor can provide assistance based on the message model information.
- When you create mappings to configure a DataDelete, DataInsert, DataUpdate, Extract, Mapping, or Warehouse node, the mapping editors can provide assistance based on the message model information.

If your application data comes from a legacy C or COBOL application, or consists of fixed-format binary data (possibly including null-terminated strings)

Use the Custom Wire Format in the MRM domain.

If your application data consists of formatted text, or contains variable length fields other than null-terminated strings

Use the Tagged Delimited String format in the MRM domain.

If your application data is created using the JMS API

Use either the XML domain, or one of the JMS domains (JMSMap and JMSStream).

If your application data is in SAP IDoc format

Use the IDOC domain.

If your application data is in MIME format, for example SOAP with attachments or RosettaNet

Use the MIME domain. You might need to parse specific parts of the message with other parsers. For example, you might parse the root of a SOAP with attachments message using the MRM XML parser.

If you do not know, or need to know, the content of your application data

Use the BLOB domain.

Partial parsing

Partial parsing is used to parse an input message bit stream only as far as is necessary to satisfy the current reference and is supported by all built-in parsers.

An input message can be of any length. To improve performance of message flows, a message is parsed only when necessary to resolve the reference to a particular part of its content. If none of the message content is referenced within the message flow (for example, the entire message is stored in a database by the Warehouse node, but no manipulation of the message content takes place), the message body is not parsed at all.

Partial parsing is implemented for the message body (containing user or application data) only. If headers are present, these are always parsed when the message is received by the input node.

Properties

This topic discusses the following types of broker properties:

- “Built-in” or broker-supplied properties, which are sometimes known simply as “broker properties”: see “Broker properties.”
- Promoted properties: see “Promoted properties.”
- User-defined properties: see “User-defined properties” on page 50.

Broker properties

For each broker, WebSphere Message Broker maintains a set of properties. You can access some of these properties from your ESQL programs. A subset of the properties is also accessible from Java code. It can be useful, during the runtime of your code, to have real-time access to details of a specific node, flow, or broker.

There are four categories of broker properties:

- Those relating to a specific node
- Those relating to nodes in general
- Those relating to a message flow
- Those relating to the execution group

“Broker properties accessible from ESQL and Java” on page 983 shows the broker, flow, and node properties that are accessible from ESQL and indicates which properties are also accessible from Java.

Broker properties:

- Are grouped by broker, execution group, flow, and node.
- Are case sensitive. Their names always start with an uppercase letter.
- Return NULL if they do not contain a value.

All nodes that allow user programs to edit ESQL support access to broker properties. These are:

- Compute nodes
- Database nodes
- Filter nodes
- All derivatives of these nodes

Promoted properties

A promoted property is a message flow node property that has been promoted to the level of the message flow in which it is included.

A message flow contains one or more message flow nodes, each of which is an instance of a message flow type (a built-in node, or a user-defined node). You can promote the properties of a message flow node to apply to the message flow to which it belongs. If you do this, any user of the message flow can set values for the properties of the nodes in this higher message flow by setting them at the message flow level, without being aware of the message flow’s internal structure.

You can promote compatible properties (that is, properties that represent comparable values) from more than one node to the same promoted property; you can then set a single property that affects multiple nodes.

For example, you might want to set the name of a data source as a property of the message flow, rather than a property of each individual node in the message flow that references that data source. You create a message flow that accesses a database

called SALESDATA. However, while you are testing the message flow, you want to use a test database called TESTDATA. If you set the data source properties of each individual node within the message flow to refer to SALESDATA, you can promote the data source property for each node in the flow that refers to it, and update the property to have the value TESTDATA which overrides the node data source properties values while you test the message flow (the promoted property always takes precedence over the settings for the properties within any relevant nodes).

A subset of message flow node properties is also configurable (that is, the properties can be updated at deploy time). You can promote configurable properties: if you do so, the promoted property (which can have a different name from the property or properties that it represents) is the one that is available to update at deploy time. Configurable properties are those associated with system resources, for example queues and data sources: they can be set at deploy time by an administrator rather than a message flow developer.

User-defined properties

A user-defined property (UDP) is a property that is defined when you construct a message flow using the Message Flow editor. This property can be used by the ESQL program inside message flow nodes such as a Compute node. It can also be used as a Java property inside the Java generated by a JavaCompute node.

The advantage of UDPs is that their values can be changed by operational staff at deployment time. You do not need to change your application programs. For example, if you use the UDPs to hold data about your computer center, you can configure a message flow for a particular machine, task, or environment at deployment time, without having to change the code at the message node level.

When you launch the Message flow editor to either create a message flow or modify an existing message flow, as well as deciding which nodes are required in the message flow, you also have the option (provided by the tab) of defining and giving initial values to some user-defined properties. Use the User Defined Properties tab at the bottom of the edit window. See Message Flow editor for more information about how you do this.

As well as being defined using the Message flow editor, a UDP must also be defined using either a DECLARE statement with the EXTERNAL keyword in any ESQL program that uses it, or the getUserDefinedAttribute method in any JavaCompute node that uses it.

See the “DECLARE statement” on page 851 for details of the DECLARE statement, and see “Accessing user-defined properties from a JavaCompute node” on page 300 for more information about how to use a UDP in a JavaCompute node.

Any value that you give to a UDP when you define it in a message flow, overrides the value of that variable in your ESQL or Java program.

The value of a UDP can also be modified at deployment time by using the Broker Archive editor to edit the bar file. This value overrides any value that was given when you defined the message flow.

Every UDP in a message flow must have a value, given either when the UDP is declared or by the Message Flow or Broker Archive editor; otherwise a deployment-time error occurs. At run time, after the UDP has been declared its value can be queried by subsequent program statements but it cannot be modified.

The value of the UDP is set at the flow level and is the same for all eligible nodes that are contained in the flow. An “eligible node” is a node that supports UDPs and is within the scope of the declaration that declares the UDP to your application. For example, if you use the Message Flow editor to change the value of a user property called “timezone”, that is declared in a schema called “mySchema”, in a message flow called “myFlow”, the UDP is available at run time to all the nodes in myFlow that support UDPs and that fall within mySchema.

Similarly, if you use the Message Flow editor to change the value of a user-defined property in a subflow, the newly edited property is available to all the nodes in the subflow that support UDPs and that are within the scope of the declaration. It is not available, for example, to nodes in the parent flow.

Message flow transactions

Message flows support two transaction styles:

1. “**Coordinated message flows**” ensure that all updates to resources are committed or rolled back together within a single transaction.
2. “**Uncoordinated message flows**” on page 52 allow updates to resources to occur independently; the updates are not affected by the overall success or failure of the flow.

Coordinated message flows

A message flow that includes interaction with an external database or other recoverable resource can be configured so that all its processing is coordinated within a single, global, transaction. This coordination ensures that either all processing is successfully completed, or no processing is completed. The transaction is committed (if all processing is successful) or rolled back (if at least one part of the processing is unsuccessful). This means that all affected resources (queues, databases, and so on) are maintained in a consistent state, and that data integrity is preserved.

Updates made by a coordinated flow are committed when the flow successfully completes processing the input message. The updates are backed out if:

1. Any node within the flow throws an exception that is not caught by a node other than the input node, *and*
2. The input node’s catch terminal is not connected.

To configure a message flow as coordinated, set the **Coordinated** property on the message flow.

For some input nodes, such as MQInput, or SCADA nodes, you can set the **Transaction Mode** property on the nodes in the flow to **Automatic**; this means that messages will be part of the global transaction, and the flow marked as transactional if the input message is persistent, and uncoordinated if the input message is not persistent. Subsequent nodes in the flow which set the transaction mode property to **Automatic** are included in the global transaction if the flow was marked transactional by the input node.

Transaction coordination of message flows is provided on distributed platforms by WebSphere MQ and on z/OS systems by RRS. Message flows are always globally coordinated on z/OS, regardless of whether the message flow’s **Coordinated** property is specified as coordinated or not.

Uncoordinated message flows

Uncoordinated flows are flows for which the **Coordinated** property is not set. Updates to resources used by a uncoordinated flow are managed by the separate resource managers. Some resource managers, such as WebSphere MQSeries, allow updates to be made non-transactionally, or as part of a resource-specific transaction. Other resource managers, such as database managers, always use a resource-specific transaction. A resource-specific transaction is a transaction whose scope is limited to the resources owned by a single resource manager, such as a database or queue manager.

Resource-specific transactions are typically used only when there is just one type of recoverable resource used in a flow. (An example of such a flow is one that contains an MQInput and an MQOutput node but which does not access any databases.) Resource-specific transactions should not be used when there is more than one resource and data integrity must be maintained.

Updates made to a resource accessed non-transactionally are committed immediately. An MQInput node configured to be non-transactional removes messages from the queue immediately, and if the flow fails the messages are lost.

Some input nodes, such as MQInput, or SCADA nodes, can be part of a transaction, depending on the persistence of the input message, by setting the transaction mode to **Automatic**. Messages are made part of the transaction, and the flow marked as transactional, if the input message is persistent, and non-transactional if the message is not persistent.

The Error Handler sample demonstrates the use of globally-coordinated transactions, and the differences in the message flow when database updates are coordinated (the main flow) and when they are not (the error flow).

Broker schemas

A broker schema is a symbol space that defines the scope of uniqueness of the names of resources defined within it. The resources are message flows, ESQ files, and mapping files.

The broker schema is defined as the relative path from the project source directory to the flow name. When you first create a message flow project, a default broker schema named (default) is created within the project.

You can create new broker schemas to provide separate symbol spaces within the same message flow project. A broker schema is implemented as a folder, or subdirectory, within the project, and provides organization within that project. You can also use project references to spread the scope of a single broker schema across multiple projects to create an application symbol space that provides a scope for all resources associated with an application suite.

A broker schema name must be a character string that starts with a Unicode character followed by zero or more Unicode characters or digits, and the underscore. You can use the period to provide a structure to the name, for example Stock.Common. A directory is created in the project directory to represent the schema, and if the schema is structured using periods, further subdirectories are defined. For example, the broker schema Stock.Common results in a directory Common within a directory Stock within the message flow project directory.

If you create a resource (for example, a message flow) in the default broker schema within a project, the file or files associated with that resource are created in the directory that represents the project. If you create a resource in another broker schema, the files are created within the schema directory.

For example, if you create a message flow Update in the default schema in the message flow project Project1, its associated files are stored in the Project1 directory. If you create another message flow in the Stock.Common broker schema within the project Project1, its associated files are created in the directory Project1\Stock\Common.

Because each broker schema represents a unique name scope, you can create two message flows that share the same name within two broker schemas. The broker schemas ensure that these two message flows are recognized as separate resources. The two message flows, despite having the same name, are considered unique.

If you move a message flow from one project to another, you can continue to use the message flow within the original project if you preserve the broker schema. If you do this, you must update the list of dependent projects for the original project by adding the target project. If, however, you do not preserve the broker schema, the flow becomes a different flow because the schema name is part of the fully qualified message flow name, and it is no longer recognized by other projects. This action results in broken links that you must manually correct. For further information about correcting errors after moving a message flow, see “Moving a message flow” on page 129.

Do not move resources by moving their associated files in the file system; you must use the workbench to move resources to ensure that all references are corrected to reflect the new organization.

The following scope and reuse conditions apply when you create functions, procedures, and constants in a broker schema:

Functions

- Functions are locally reusable and can be called by module-scope subroutines or mappings within the same schema.
- Functions are globally reusable and can be called by other functions or procedures in ESQL or mapping files within any schema defined in the same or another project.

Procedures

- Procedures are locally reusable and can be called from module-scope subroutines in ESQL files within the same schema.
- Procedures are globally reusable and can be called by other functions or procedures in ESQL files within any schema defined in the same or another project.

Procedures cannot be used in mapping files.

Constants

- Constants are locally reusable and can be used where they are defined in any ESQL or mapping file within the same broker schema.
- Constants are not globally reusable; you cannot use a constant that is declared in another schema.

If you want to reuse functions or procedures globally:

- Specify the path of the function or procedure:
 - If you want to reuse a function or procedure in an ESQL file, either provide a fully-qualified reference, or include a PATH statement that defines the path. If you choose to define the path, code the PATH statement in the same ESQL file as that in which the function is coded, but not within any MODULE.
 - If you want to reuse a function in a mapping file, do one of the following:
 - Qualify the function in the Composition Expression editor.
 - Select **Organize Schema References** in the outline view. This detects dependent PATHs and automatically updates the reference.
 - Select **Modify Schema References** in the outline view. You can then select the schema in which the function is defined.

(You cannot reuse a procedure in a mapping file.)
- Set up references between the projects in which the functions and procedures are defined and used.

Message flow accounting and statistics data

Message flow accounting and statistics data is the information that can be collected by a broker to record performance and operating details of message flow execution.

These reports are not the same as the publish/subscribe statistics reports that you can generate. The publish/subscribe statistics provide information about the performance of brokers, and the throughput between the broker and clients that are connected to the broker. Message flow accounting and statistics reports provide information about the performance and operating details of a message flow execution.

Message flow accounting and statistics data records dynamic information about the runtime behavior of a message flow. For example, it indicates how many messages are processed and how large those messages are, as well as CPU usage and elapsed processing times. The broker collects the data and records it in a specified location when one of a number of events occurs (for example, when a snapshot interval expires or when the execution group you are recording information about stops).

Accounting and statistics data is collected only for message flows that start with an MQInput, HTTPInput, or user-defined input node. If you start data collection for a message flow that starts with one of these nodes, the data is collected for all built-in and user-defined nodes, including those in subflows. If the message flow starts with another input node (for example, a Real-timeInput node), no data is collected (and no error is reported).

Collecting message flow accounting and statistics data is optional; by default it is switched off. To use this facility, request it on a message flow or execution group basis. The settings for accounting and statistics data collection are reset to the defaults when an execution group is redeployed. Previous settings for message flows in an execution group will not be passed on to the new message flows deployed to that execution group. Data collection is started and stopped dynamically when you issue the `mqsichangeflowstats` command; you do not need to make any change to the broker or to the message flow, or redeploy the message flow, to request statistics collection.

You can activate data collection on both your production and test systems. If you collect the default level of statistics (message flow), the impact on broker performance is minimal. However, collecting more data than the default message flow statistics can generate high volumes of report data that might cause a small but noticeable performance overhead.

When you plan data collection, consider the following points:

- Collection options
- Accounting origin
- Output formats

You can find more information on how to use accounting and statistics data to improve the performance of a message flow in this developerWorks article on message flow performance.

The following SupportPac provides additional information about using accounting and statistics:

- Using statistics and accounting SupportPac (IS11)

Message flow accounting and statistics collection options

The options that you specify for message flow accounting and statistics collection determine what information is collected. You can request the following types of data collection:

- Snapshot data is collected for an interval of approximately 20 seconds. The exact length of the interval depends on system loading and the level of current broker activity. You cannot modify the length of time for which snapshot data is collected. At the end of this interval, the recorded statistics are written to the output destination and the interval is restarted.
- Archive data is collected for an interval that you have set for the broker on the `mqscreatebroker` or `mqschangebroker` command. You can specify an interval of between 10 and 14400 minutes, the default value is 60 minutes. At the end of this interval, the recorded statistics are written to the output destination and the interval is restarted.

An interval is prematurely expired and restarted when any of the following events occur:

- The message flow is redeployed.
- The set of statistics data to be collected is modified.
- The broker is shut down.

This preserves the integrity of the data already collected when that event occurs.

On z/OS, you can set the command parameter to 0, which means that the interval is controlled by an external timer mechanism. This support is provided by the Event Notification Facility (ENF), which you can use instead of the broker command parameter if you want to coordinate the expiration of this timer with other system events.

You can request snapshot data collection, archive data collection, or both. You can activate snapshot data collection while archive data collection is active. The data recorded in both reports is the same, but is collected for different intervals. If you activate both snapshot and archive data collection, be careful not to combine information from the two different reports, because you might count information twice.

You can use the statistics generated for the following purposes:

- You can record the load that applications, trading partners, or other users put on the broker. This allows you to record the relative use that different users make of

the broker, and perhaps to charge them accordingly. For example, you could levy a nominal charge on every message that is processed by a broker, or by a specific message flow.

Archive data provides the information that you need for a use assessment of this kind.

- You can assess the execution of a message flow to determine why it, or a node within it, is not performing as you expect.

Snapshot data is appropriate for performance assessment.

- You can determine the route that messages are taking through a message flow. For example, you might find that an error path is taken more frequently than you expect and you can use the statistics to understand when the messages are routed to this error path.

Check the information provided by snapshot data for routing information; if this is insufficient for your needs, use archive data.

Message flow accounting and statistics accounting origin

Accounting and statistics data can be accumulated and reported with reference to an identifier associated with a message within a message flow. This identifier is the accounting origin. This provides a method of producing individual accounting and statistics data for multiple accounting origins that generate input to message flows. The accounting origin can be a fixed value, or it can be dynamically set according to your criteria.

For example, if your broker hosts a set of message flows associated with a particular client in a single execution group, you can set a specific value for the accounting origin for all these flows. You can then analyze the output provided to assess the use that the client or department makes of the broker, and charge them accordingly.

If you want to track the behavior of a particular message flow, you can set a unique accounting origin for this message flow, and analyze its activity over a given period.

To make use of the accounting origin, you must perform the following tasks:

- Activate data collection, specifying the correct parameter to request basic support (the default is none, or no support). For details, see `mqsichangeflowstats` command.
- Configure each message flow for which you want a specific origin to include ESQL statements that set the unique value that is to be associated with the data collected. Data for message flows for which a specific value has not been set are identified with the value `Anonymous`.

The ESQL statements can be coded in a Compute, Database, or Filter node.

You can configure the message flow either to set a fixed value, or to determine a dynamic value, and can therefore create a very flexible method of recording sets of data that are specific to particular messages or circumstances. For more information, refer to “Setting message flow accounting and statistics accounting origin” on page 376.

You can complete these tasks in either order; if you configure the message flow before starting data collection, the broker ignores the setting. If you start data collection, specifying accounting origin support, before configuring the message flow, all data is collected with the Accounting Origin set to `Anonymous`. The broker acknowledges the origin when you redeploy the message flow. You can also modify data collection that has already started to request accounting origin

support from the time that you issue the command. In both cases, data that has already been collected is written out and collection is restarted.

When data has been collected, you can review information for one or more specific origins. For example, if you select XML publication messages as your output format, you can start an application that subscribes to the origin in which you are interested.

Output formats for message flow accounting and statistics data

When you collect message flow statistics, you can choose the output destination for the data:

- User trace
- XML publication
- SMF

Statistics data is written to the specified output location in the following circumstances:

- When the archive data interval expires.
- When the snapshot interval expires.
- When the broker shuts down. Any data that has been collected by the broker, but not yet written to the specified output destination, is written during shutdown. It might therefore represent data for an incomplete interval.
- When any part of the broker configuration is redeployed. Redeployed configuration data might contain an updated configuration that is not consistent with the existing record structure (for example, a message flow might include an additional node, or an execution group might include a new message flow). Therefore the current data, which might represent an incomplete interval, is written to the output destination. Data collection continues for the redeployed configuration until you change data collection parameters or stop data collection.
- When data collection parameters are modified. If you update the parameters that you have set for data collection, all data collected for the message flow (or message flows) is written to the output destination to retain data integrity. Statistics collection is restarted according to the new parameters.
- When an error occurs that terminates data collection. You must restart data collection yourself in this case.

User trace

You can specify that the data collected is written to the user trace log. The data is written even if trace is currently switched off. The default output destination for accounting and statistics data is the user trace log. The data is written to one of the following locations:

- On Windows systems, if the broker workpath has been set using the `-w` option of the `mqsicreatebroker` command, data is written to `workpath\log`. If the broker workpath has not been specified, data is written to `install_dir\log`, where `install_dir` is the directory in which WebSphere Message Broker is installed.
- On UNIX systems, data is written to `/var/wmqi/log`.
- On Linux systems, data is written to `/var/wmqi/log`.
- On z/OS systems, data is written to `/component_filesystem/log`.

XML publication

You can specify that the data collected is published. The publication message is created in XML format and is available to subscribers registered in the broker network that subscribe to the correct topic.

The topic on which the data is published has the following structure:

```
$SYS/Broker/brokerName/StatisticsAccounting/recordType/executionGroupLabel/messageFlowLabel
```

The variables correspond to the following values:

brokerName

The name of the broker for which statistics are collected.

recordType

Set to Snapshot or Archive, depending on the type of data to which you are subscribing. Alternatively, use # to register for both snapshot and archive data if it is being produced.

executionGroupLabel

The name of the execution group for which statistics are collected.

messageFlowLabel

The label on the message flow for which statistics are collected.

Subscribers can include filter expressions to limit the publications that they receive. For example, they can choose to see only snapshot data, or to see data collected for a single broker. Subscribers can specify wild cards (+ and #) to receive publications that refer to multiple resources.

The following examples show the topic with which a subscriber should register to receive different sorts of data:

- Register the following topic for the subscriber to receive data for all message flows running on *BrokerA*:
\$SYS/Broker/*BrokerA*/StatisticsAccounting/#
- Register the following topic for the subscriber to receive only archive statistics relating to a message flow *Flow1* running on execution group *Execution* on broker *BrokerA*:
\$SYS/Broker/*BrokerA*/StatisticsAccounting/Archive/*Execution*/*Flow1*
- Register the following topic for the subscriber to receive both snapshot and archive data for message flow *Flow1* running on execution group *Execution* on broker *BrokerA*:
\$SYS/Broker/*BrokerA*/StatisticsAccounting/#/*Execution*/*Flow1*

Message display, test and performance utilities SupportPac (IH03) can help you with registering your subscriber.

SMF

On z/OS, you can specify that the data collected is written to SMF. SMF supports the collection of data from multiple subsystems, and you might therefore be able to synchronize the information recorded. When you want to interpret the information recorded, you can use any utility program that processes SMF records. Accounting and statistics data uses SMF type 117 records.

Designing a message flow

When you design a message flow, consider several design factors which include some or all of the following options:

- Which nodes provide the function that you require. In many cases, you can choose between several nodes that provide a suitable function. You might have to consider other factors listed here to determine which node is best for your overall needs. You can include built-in nodes, user-defined nodes, and subflow nodes. For more information, see “Deciding which nodes to use” on page 60.
- Whether it is appropriate to include more than one input node. For more information, see “Using more than one input node” on page 68.
- How to specify the characteristics of the input message. See “Defining input message characteristics” on page 68 for more details.
- Whether you want to determine the path that a message follows through the message flow based on the content or the characteristics of the message. Several nodes provide checks or examination of the message and output terminals that can be connected to direct certain messages to different nodes. This is further described in “Using nodes for decision making” on page 69.
- Whether you can make use of subflows that provide a well-defined subset of processing. You might be able to reuse subflows created for another project (for example, an error processing routine). Or you might want to create a subflow in your current project and reuse it in several places within the same message flow. For more information, see “Using subflows” on page 71.
- What response times your applications expect from the message flow. This is influenced by several aspects of how you configure your nodes and the flow. For more information, see “Optimizing message flow response times” on page 73.
- Whether you can use the destination list within the LocalEnvironment associated with the message to determine the processing within the message flow (using RouteToLabel and Label nodes) or the target for the output messages (for example, by setting the *Destination Mode* property of the MQOutput node to Destination List). For more information, see “Creating destination lists” on page 76.
- Whether you want to use WebSphere MQ cluster queues. For more information, see “Using WebSphere MQ cluster queues for input and output” on page 77.
- Whether you want to use WebSphere MQ shared queues on z/OS. Their use is described further in “Using WebSphere MQ shared queues for input and output (z/OS)” on page 78.
- Whether you want to validate input messages received by the input node, or output messages generated by the Compute node, or both. For more information, see “Validating messages” on page 79.
- Whether you want to view or record message structure in Trace node output. Information about how you can do this is provided in “Viewing the logical message tree in trace output” on page 81.
- Whether your message flows access data in databases. You must configure message flow nodes, databases, and database connections to enable this, as described in “Accessing databases from message flows” on page 84.
- Whether your messages must be handled within a transaction. Some built-in nodes have properties that you can set to control how transactions are managed, and how messages are processed within a transaction. For more information, see “Configuring coordinated message flows” on page 93.

- Whether you want your messages to go through data conversion. The options that you have are described in “Configuring message flows for data conversion” on page 107.
- What steps you can take to ensure that messages are not lost. For more information, see “Ensuring that messages are not lost” on page 109.
- How errors are handled within the message flow. You can use the facilities provided by the broker to handle any errors that are encountered during message flow execution (for example, if the input node fails to retrieve an input message, or if writing to a database results in an error). However, you might prefer to design your message flow to handle errors in a specific way. For more information, see “Handling errors in message flows” on page 111.

Deciding which nodes to use

WebSphere Message Broker includes a large number of message processing nodes that you can use within your message flows. It also provides an interface that you can use to define your own nodes, known as user-defined nodes.

Your decision about which nodes to use depends on the processing that you want to perform on your messages. The built-in nodes can be considered in several categories, and are displayed in the workbench grouped in those categories (although this grouping has no effect on their operation). You can also categorize user-defined nodes in the same way. The categories are:

Input and output

Input and output nodes define points in the message flow to which clients send messages (input nodes such as MQInput) and from which clients receive messages (output nodes such as MQOutput). Client applications interact with these nodes by putting messages to, or getting messages from, the I/O resource that is specified by the node as the source or target of the messages. Although a message flow must include at least one input node, it does not have to include an output node.

- If you are creating a message flow that you want to deploy to a broker, you must include at least one input node to receive messages. The input node that you choose depends on the source of the input messages and where in the flow you want to receive the messages:

MQInput

If the messages arrive at the broker on a WebSphere MQ queue and the node is to be at the start of a message flow.

MQGet

If the messages arrive at the broker on a WebSphere MQ queue and the node is not to be at the start of a message flow.

SCADAInput

If the messages are sent by a telemetry device.

HTTPInput

If the messages are sent by a Web services client.

Real-timeInput or Real-timeOptimizedFlow

If the messages are sent by a JMS or multicast application.

User-defined input node

If the message source is a client or application that uses a different protocol or transport.

Input node

If you are creating a message flow that you want to embed in another message flow (a subflow) that you will not deploy as a standalone message flow, you must include at least one Input node to receive messages into the subflow.

An instance of the Input node represents an in terminal. For example, if you have included one instance of the Input node, the subflow icon shows one in terminal that you can connect to other nodes in the main flow in the same way that you connect any other node.

You can deploy only message flows that have at least one input node. If your message flow does not contain an input node, you are prevented from adding it to the broker archive file. The input node can be in the main flow, or in a message flow that is embedded in the main flow.

You can use more than one input node in a message flow. For more information, see “Using more than one input node” on page 68.

- If you want to send the messages produced by the message flow to a target application, you can include one or more output nodes. The one that you choose depends on the transport across which the target application expects to receive those messages:

Publication

If you want to distribute the messages using the publish/subscribe network for applications that subscribe to the broker across all supported protocols. A Publication node is an output node that use output destinations that are identified by subscribers whose subscriptions match the characteristics of the current message.

MQOutput

If the target application expects to receive messages on a WebSphere MQ queue, or on the WebSphere MQ reply-to queue specified in the input message MQMD.

MQReply

If the target application expects to receive messages on the WebSphere MQ reply-to queue specified in the input message MQMD

SCADAOutput

If a telemetry device is the target of the output messages, and the Publication node is not suitable

HTTPReply

If the messages are for a Web services client

HTTPRequest

If your message flow interacts with a Web service client

Real-timeOptimizedFlow

If the target application is a JMS or multicast application

User-defined output node

If the target is a client or application that uses a different protocol or transport

Output node

If you are creating a message flow that you want to embed in

another message flow (a subflow) that you will not deploy as a standalone message flow, you must include at least one Output node to propagate messages to subsequent nodes that you connect to the subflow.

An instance of the Output node represents an out terminal. For example, if you have included two instances of the Output node, the subflow icon shows two out terminals that you can connect to other nodes in the main flow in the same way that you connect any other node.

Message manipulation, enhancement, and transformation

Most enterprises have applications that have been developed over many years, on different systems, using different programming languages, and different methods of communication. WebSphere Message Broker removes the need for applications to understand these differences because it provides the ability to configure message flows that transform messages from one format to another.

For example, personal names are held in many forms in different applications. Surname first or last, with or without middle initials, upper or lower case: these are just some of the permutations. Because you can configure your message flow to know the requirements of each application, each message can be transformed to the correct format without modifying the sending or receiving application.

You can work with the content of the message to update it in several ways. Your choices here might depend on whether the message flow must handle predefined (modelled) messages, or self-defining messages (for example, XML), or both.

A message flow can completely rebuild a message, convert it from one format to another (whether format means order of fields, byte order, language, and so on), remove content from the message, or introduce specific data into it. For example, a node can interact with a database to retrieve additional information, or to store a copy of the message (whole or part) in the database for offline processing.

The following examples show how important message transformation can be:

- An order entry application has a Part ID in the body of the message, but its partner stock application expects it in the message header. The message is directed to a message flow that knows the two different formats, and can therefore reformat the information as it is needed.
- A data-entry application creates messages containing stock trade information. Some applications receiving this message need the information as provided, but others need additional information added to the message about the price to earnings (PE) ratio. The stock trade messages are directed to a message flow that passes the message unchanged to some output nodes, but calculates and adds the extra information for the others. The message flow does this by looking up the current stock price in a database, and uses this value and the trade information in the original message to calculate the PE value before passing on the updated message.

You can also create message flows that interact with each other using these nodes. Although the default operation of one message flow does not

influence the operation of another message flow, you can force this by configuring your message flows to store and retrieve information in an external source such as a database.

Compute

You can manipulate message content, transform the message in some way, and interact with a database to modify the content of the message or the database content and pass on one or more new messages using the Compute node. You can use this node to manipulate predefined and self-defining messages.

Use the ESQL editor to create an ESQL module, specific to this node, that contains the statements defining the actions to perform against the message or database. Do not use the ESQL code that you develop for use in a Compute node in any other type of node.

You can control the way in which the database is accessed by this node by specifying user and password information for the datasource that you specify in the node property. Use the `mqsisetdbparms` command to initialize and maintain these values.

You can control the way in which the database is accessed by this node by specifying user and password information for the datasource that you specify in the node property. Use the `mqsisetdbparms` command to initialize and maintain these values.

If your message manipulation requirements are complex, complete these within a single Compute node. Fewer, more complex Compute nodes perform better than a larger number of simpler nodes, because the broker parses the message on entry to each Compute node.

Mapping

You can create a new message from the input message by mapping the content of elements of the output message from elements of the input message, or from database content, using the Mapping node. You can also extract parts of the message, and optionally change their content, to create a new output message that is a partial copy of the message received by the node. The Mapping node handles only predefined messages.

You can control the way in which the database is accessed by this node by specifying user and password information for the datasource that you specify in the node property. Use the `mqsisetdbparms` command to initialize and maintain these values.

Using a mapping editor, you can develop mappings to perform simple manipulations on predefined messages in the Mapping node. Do not use the mappings that you develop for use in a Mapping node in any other type of node.

Extract

You can use the Extract node to create a new output message from specified elements of the input message. You can extract parts of the message, and optionally change their content, to create a new output message that is a partial copy of the message received by the node. The Extract node handles only predefined messages.

Using a mapping editor, you can develop mappings to perform simple manipulations on predefined messages in the Extract node. Do not use the mappings that you develop for use in an Extract node in any other type of node.

Database

Use the Database node to interact with a database identified by the node properties. The Database node handles both predefined and self-defining messages. Using an ESQL editor, you can code ESQL functions to update database content from the message, insert new information into the database, and delete information from the database, perhaps based on information in the message. Do not use the ESQL code that you develop for use in a Database node in any other type of node.

This node provides a very flexible interface with a wide range of functions. It also has properties that you can use to control the way in which the interaction participates in transactions.

You can control the way in which the database is accessed by this node by specifying user and password information for the datasource that you specify in the node property. Use the `mqsisetdbparms` command to initialize and maintain these values.

You can only update a database from this node; you cannot update any message content. If you want to update message content, use the Compute or Mapping node.

DataDelete, DataInsert, DataUpdate

The DataDelete, DataInsert, and DataUpdate nodes are specialized forms of the Database node that provide a single mode of interaction (deletion of one or more rows, insertion of one or more rows, or update of one of more existing rows respectively). The DataDelete, DataInsert, and DataUpdate nodes handle only predefined messages. Use a mapping editor to develop mappings to perform these functions. Do not use the mappings that you develop for these nodes in any other type of node. These nodes also let you control the transactional characteristics of the updates that they perform.

You can control the way in which the database is accessed by this node by specifying user and password information for the datasource that you specify in the node property. Use the `mqsisetdbparms` command to initialize and maintain these values.

You can only update databases from these nodes: you cannot update any message content. If you want to update message content, use the Compute or Mapping node.

Warehouse

The Warehouse node provides a store interface that you can use to store all or part of the message in a database, for example, for audit reasons. The Warehouse node handles only predefined messages. Use a mapping editor to develop mappings to perform this action. Do not use the mappings that you develop for a Warehouse node in any other type of node.

You can control the way in which the database is accessed by this node by specifying user and password information for the

datasource that you specify in the node property. Use the `mqsisetdbparms` command to initialize and maintain these values.

You can only update a database from this node; you cannot update any message content. If you want to update message content, use the Compute or Mapping node.

User-defined

You can include a user-defined node to interact with a database.

You can control the way in which the database is accessed by this node by specifying user and password information for the datasource that you specify in the node property. Use the `mqsisetdbparms` command to initialize and maintain these values.

The following table summarizes what you can update in these nodes.

Node	Update database?	Update message?	Update LocalEnvironment?	Message set required?
Compute	Yes	Yes	Yes	No
Database	Yes	No	Yes	No
DataDelete	Yes	No	Yes	Yes
DataInsert	Yes	No	Yes	Yes
DataUpdate	Yes	No	Yes	Yes
Extract	Yes	Yes	Yes	Yes
Mapping	Yes	Yes	Yes	Yes
Warehouse	Yes	No	Yes	Yes

XMLTransformation

If you want to transform an input XML message into another format using XMLT style sheets, use the XMLTransformation node. It is imperative that the data can be parsed into a XML message. The result of the transformation is output as a BLOB message. The style sheet, using the rules defined within it, can sort the data; select data elements to include or to exclude based on some criteria, and transform the data into some other data format.

The Xalan-Java transformation engine (<http://xml.apache.org/xalan-j>) is used as the underlying transformation engine. For details about XMLT, refer to <http://www.w3.org/TR/xslt>.

You can deploy style sheets and XML files to broker execution groups, to facilitate style sheet and XML file maintenance.

Decision making

You can use nodes that determine the order and flow of control within the message flow in various ways to make decisions about how messages are processed by the flow. You can also use nodes (TimeoutControl and TimeoutNotification) that determine the time, and frequency of occurrence, of events within the message flow. Routing is independent of message transformation, although the route a message takes might determine exactly what transformation is performed on it.

For example, a money transfer application always sends messages to one other application. You might decide that every message with a transfer

value of more than \$10,000 must also be sent to a second application, to enable all high-value transactions to be recorded.

In another example, a national auto club offers a premier service to specific members for orders above a threshold value. Most orders are routed through the usual channels, but, if the membership number and order value meet certain criteria, the order gets special treatment.

You can also establish a more dynamic routing option by building additional routing information into the message when it is processed. Optional sets of rules are set up to receive messages according to values (destinations) set into the message. You can establish these rules such that a message is processed by one or more of the optional sets of rules, in an order determined by the added message content.

Use the following nodes to make decisions about the route that a message follows through the message flow:

Check You can inspect the message characteristics to determine its structure, and route the message according to whether it meets the required characteristics.

Filter You can code an ESQL statement to determine the next node to which the message is sent by this node. Do not use the ESQL code that you develop for use in a Filter node in any other type of node.

The node's terminals are true, false, unknown, and failure; the message is propagated to the true terminal if the test succeeds, and to the false terminal if it fails. If the statement cannot be resolved (for example, it tests the value of a field that does not appear in the input message), the message is propagated to the unknown terminal. If any other error is detected, the message is propagated to the failure terminal.

The test in the ESQL statement can depend on message content, database content, or a combination of these.

If you reference a database, you can control the way in which it is accessed by this node by specifying user and password information for each datasource defined in the registry on the broker system. On distributed systems, use the `mqsisetdbparms` command to initialize and maintain these values.

On z/OS, use the broker started task ID to access the database.

Use this node in preference to the Compute node to provide this function; although you can configure the Compute node to control message selection and routing, the Filter node performs better.

FlowOrder

You can connect the terminals of this node to force the message to be processed by one sequence of nodes, followed by a second sequence of nodes.

Passthru

Use the Passthrough node in a subflow as the first node that follows the Input node to identify the subflow in which it is included. For example, if you develop an error processing subflow to include in several message flows, you might want to modify that subflow. However, you might want to introduce the modified version initially to just a subset of the message flows in which it is

included. Set a value for the instance of the Passthrough node that identifies which version of the subflow you have included.

RouteToLabel and Label

You can define a list of destinations in a Compute node that are acted on by the RouteToLabel node, which interrogates the destinations and passes the message on to the corresponding Label node.

ResetContentDescriptor

You can set new message properties that are used when the message bit stream is next parsed by a subsequent node in the message flow.

You might want a batch job to run every day at a specific time, or you might want information to be processed and published at fixed intervals (for example, currency exchange rates are calculated and sent to banks), or you might want to take a specified recovery action if certain transactions are not completed within a defined time. For all these cases two timeout nodes (TimeoutControl and TimeoutNotification) are provided.

TimeoutControl

Use a TimeoutControl node and a TimeoutNotification node together in a message flow to control events that occur at a specific time or at defined time intervals. The TimeoutControl node receives an input message that contains a timeout request. This input message is validated and stored (all or a part of the message) to be propagated by an associated TimeoutNotification node in the message flow. The input message is also propagated unchanged to the next node in the message flow.

Note: More than one TimeoutControl node can be associated with each TimeoutNotification node.

TimeoutNotification

Use a standalone TimeoutNotification node to generate messages that are propagated at configured times or time intervals to the next node in the message flow for further processing.

Collating requests

You can collate related requests and responses using the AggregateControl, AggregateReply, and AggregateRequest nodes. Use these nodes to generate several requests in response to one input message, and to control and coordinate the responses that are received in response to those requests, and to combine the information provided by the responses to continue processing.

Error handling and reporting

You can use nodes that affect error handling and reporting:

Trace When you include a Trace node, you can generate one or more trace entries to record what is happening in the message flow at this point.

TryCatch

When you include a TryCatch, you can control the error processing when exceptions are thrown.

Throw

When you include a Throw node, you can force an exception to be thrown, and specify the identity of the exception, to make it easier to diagnose the problem.

With the exception of the Compute, Extract, and Mapping nodes, the input message received by a node and the output message sent on by the node are identical.

Using more than one input node

You can include more than one input node in a single message flow. You might find this useful in the following situations:

- The message flow provides common processing for messages that are received across multiple transports. For example, a single message flow might handle:
 - Data in messages received across WebSphere MQ, and therefore through a WebSphere MQ queue and an MQInput node
 - Messages that are received across native IP connections (a Real-timeInput node)
- You need to set standard properties on the MQInput node if input messages:
 - are predefined, and
 - are all received across WebSphere MQ, and
 - do not include an MQRFH2 header.

If the required standard properties are not always the same for every message, you can include more than one input node and configure each to handle a particular set of properties.

This is not required for self-defining messages.

- Each input node in a message flow causes the broker to start a separate thread of execution. Including more than one input node might improve the message flow performance. However, if you include multiple input nodes that access the same input source (for example, a WebSphere MQ queue), the order in which the messages are processed cannot be guaranteed. If you want the message flow to process messages in the order in which they are received, this option is not appropriate.

If you are not concerned about message order, consider using additional instances of the same message flow rather than multiple input nodes. If you set the *Additional Instances* property of the message flow when you deploy it to the broker, multiple copies of the message flow are started in the execution group. This is the most efficient way of handling multiple instances.

The Scribble sample uses two input nodes: an MQInput node and a Real-timeInput node. This enables the sample's message flow to accept input across both WebSphere MQ transport and native IP connections.

Defining input message characteristics

When a message is received by an input node in a message flow, the node detects how to interpret that message by determining the domain in which the message is defined and invoking the appropriate parser.

You can provide message domain information to the input node in one of two ways:

1. You can configure the built-in input nodes to indicate the message domain, and therefore the parser to be invoked, for each message that it receives.
2. You can set values in the input message itself that specify this information. Do this with the MQRFH2 header, which contains a folder that defines the message characteristics. This is a more flexible approach, because it means that the input node can make the decision based on the content of each message.

If the input message is defined in the MRM domain, and is therefore interpreted by the MRM parser, you must specify the following additional properties:

- *Message Set* within which the message is defined
- *Message Type* defined by the message model
- *Message Format*, the physical characteristics of the message

If the message is a WebSphere MQ message, these properties can be set either in the input node or in the MQRFH2 header of the incoming message (if they are set in both, the contents of the MQRFH2 header take precedence).

If the input message belongs to a message domain other than those for which a parser is supplied, you must provide a user-defined parser to handle it, and a user-defined input node to accept it for processing in the message flow. Check the documentation provided with the user-defined parser and node for further information.

If the input node cannot determine the message characteristics, the message is considered to be in the BLOB domain, and the BLOB parser is invoked.

Import the Video Rental sample or the Comma Separated Value (CSV) sample (or another sample that uses a message set) from the Samples Gallery and look at the values on the Default properties page of the input node in the sample's message flow.

Using nodes for decision making

You can use several built-in nodes in different ways to control the path that a message takes through the message flow.

These nodes let you decide how messages are processed by specifying the route that each message takes through the message flow based on dynamic values such as message structure and content.

For more information, see the following topics:

- "Testing the message structure (Check node)"
- "Controlling the order of processing within a message flow" on page 70
- "Testing the message content (Filter node)" on page 70
- "Using the destination list to route messages (RouteToLabel and Label nodes)" on page 71

Testing the message structure (Check node)

Use the Check node to test the characteristics of the message structure.

If you set the node properties appropriately, you can request that one or all of the message domain, message set, and message type are compared to a specific value. If the message matches those values for which you have requested the check, it is routed through the match terminal and is processed by the sequence of nodes that you have connected to that terminal.

If the message does not match any one of those values for which you have requested the check, it is routed through the failure terminal and is processed by the sequence of nodes that you have connected to that terminal.

For example, you might design a message flow that provides additional processing for all messages that are in the MRM domain. You can include a Check node that tests just that characteristic of the message, and passes it to a sequence of nodes that provide the specialized processing. If the message is not in the MRM domain, the extra nodes are bypassed, and the failure terminal is wired up directly to the node that follows the sequence required for MRM messages only.

Controlling the order of processing within a message flow

Use the FlowOrder node to control the order of processing within a message flow.

When you connect message flow nodes together, the broker determines the way in which the different connections are processed. This includes the order in which they are processed. If you have connected more than one node or sequence of nodes to a single output terminal, you cannot predict whether one sequence is processed before another for any given message.

If the order of processing is important in your message flow, use the FlowOrder node to force a prescribed order of processing of the messages that are propagated by this node.

The FlowOrder node has two output terminals that you can connect to control the order in which subsequent nodes process the message. The output terminals, named *first* and *second*, are always processed in that order.

When you connect a node or sequence of nodes to the terminal named *first*, the input message is passed to the next node, and all processing defined by all subsequent nodes in this sequence is completed before control returns to the FlowOrder node.

The input message is then propagated to the next node in the sequence of nodes connected to the terminal named *second*.

The message passed to both sequences of nodes, from the terminal named *first* and the terminal named *second*, is identical. It is always the message that the FlowOrder node receives as input. The message that the FlowOrder node propagates to the terminal named *second* is in no way affected by the processing of the message that has been performed by the sequence of nodes connected to the terminal named *first*.

The FlowOrder node provides no other processing on the input message; it is used only for imposing order on subsequent processing.

Testing the message content (Filter node)

Use the Filter node to determine the path taken by a message through the message flow based on its content.

You can customize the Filter node using ESQL statements to determine if the message content meets some condition. The condition tested must yield a Boolean result, that is it must be true or false (or unknown). You can create the test to reference information from a database, if applicable.

You can connect nodes following the Filter node to the corresponding terminals of the Filter node, and process the message according to its content.

Look at the following samples to see how to use the Filter node:

- Airline
- Error Handler

Using the destination list to route messages (RouteToLabel and Label nodes)

You can determine the path that a message takes through the message flow using the RouteToLabel and Label nodes. These provide a more flexible way to process messages than the Filter node, which depends on the Boolean result of an ESQL expression for its logic.

When you use RouteToLabel and Label, you must include a Compute node that determines, using some combination of message content, database content, and ESQL logic, how messages are to be processed next. Configure the Compute node to create a destination list (within the DestinationList folder in the LocalEnvironment subtree) that contains the destination for each message, specified as the LabelName of a Label node. The Compute node passes the message to the RouteToLabel node, which reads the destination list and propagates the message to those destinations. You can configure the RouteToLabel node to work through the destinations from first to last, or last to first. There is no limit to the number of destinations that the Compute node writes in the destination list.

If you intend to derive destination values from the message itself, or from a database, you might also need to cast values from one type to another. For more information about the LocalEnvironment, see “LocalEnvironment tree” on page 18. For more information about casting, see “Supported casts” on page 965.

Look at the following samples to see how to use these nodes:

- Airline Reservations sample

This use of the destination list is in contrast to its use to define the final recipients of the output messages; this is described in “Creating destination lists” on page 76.

The XML_PassengerQuery message flow in the Airline Reservations sample demonstrates how you can use the destination list in the LocalEnvironment to route messages based on the information in the message itself.

Using subflows

You can include subflows in your message flows in exactly the same way as you include built-in or user-defined nodes.

You can also connect them to other nodes in the same way. Because you can define a subflow once, and use it in more than one message flow (and even in more than one message flow project), a subflow can provide benefits:

- Reuse and reduced development time.
- A consistent way of achieving a particular function, and increased maintainability of your message flows (consider a subflow as analogous to a programming macro, or to inline code that is written once but used in many places).

- Flexibility. If you promote some or all of the properties of the nodes in the subflow, you can tailor a subflow to a specific context (for example, by updating the output queue or data source information).

However, you must remember that a subflow is not a single node, and its inclusion increases the number of nodes in the message flow, which might affect its performance.

Consider these examples of subflow use:

- You can define a subflow that provides a common sequence of actions that applies to several message flows if an error is encountered. For example, you might have a common error routine that writes the message to a database through the Warehouse node, and puts it to a queue for processing by an error recovery routine. The use of this routine in multiple message flows, or in several places within one message flow, provides an efficient and consistent use of resources and avoids reinventing such routines every time an error is encountered.
- You might have a common calculation that you want to perform on messages that pass through several different message flows. For example, you might access currency exchange rates from a database and apply these to calculate prices in several different currencies. You can include the currency calculator subflow in each of the message flows in which it is appropriate.

You can use the Passthrough node in a subflow as the first node that follows the Input node to identify the subflow in which it is included. You can specify an identifier (Label) in whatever way meets your requirements, for example to identify the level or version of the flow in which it is configured. The Passthrough node does not process the message in any way. The message that it propagates on its out terminal is the same message that it received on its in terminal. For example, if you develop an error processing subflow to include in several message flows, you might want to modify that subflow. However, you might want to introduce the modified version initially to just a subset of the message flows in which it is included. Set a value for the instance of the Passthrough node that identifies which version of the subflow you have included.

The use of subflows is illustrated in the following sample:

- Error Handler sample

The use of subflows is demonstrated in the Error Handler sample and the Coordinated Request Reply sample. The Error Handler sample uses a subflow to trap information about errors and store the information in a database. The Coordinated Request Reply sample uses a subflow to encapsulate the storage of the ReplyToQ and ReplyToQMgr values in a WebSphere MQ message so that the processing logic can be easily reused in other message flows and to allow alternative implementations to be substituted.

Adding keywords to subflows

You can embed keywords in each subflow that you use in a message flow. A different keyword must be used in each instance of a subflow. This is because only the first recorded instance of each keyword within the message flow .cmf file is available to Configuration Manager Proxy applications and to the toolkit. The order that subflows appear in the .cmf file is not guaranteed.

Optimizing message flow response times

When you design a message flow, the flexibility and richness of the built-in nodes often means that there are several ways to achieve the processing and therefore the end results that you require. However, you can also find that these different solutions deliver different performance and, if this is an important consideration, you must design for performance as well as function.

There are two ways in which your applications can perceive performance:

1. **Response time.** This indicates how quickly each message is processed by the message flow. This is particularly influenced by how you design your message flows. This is further discussed in this topic.
2. **Throughput.** This indicates how many messages of particular sizes can be processed by a message flow in a given time. This is mainly affected by configuration and system resource factors, and is therefore discussed in *Optimizing message flow throughput with other domain configuration information*.

There are several aspects that influence message flow response times. However, as you create and modify your message flow design to arrive at the best results that meet your specific business requirements, you must also consider the eventual complexity of the message flow. The most efficient message flows are not necessarily the easiest to understand and maintain; experiment with the solutions available to arrive at the best balance for your needs.

Several factors influence message flow response times:

The number of nodes that you include in the message flow

Every node causes some processing overhead, so consider the content of the message flow carefully, including the use of subflows.

Use as few nodes as possible in a message flow; every node that you include in the message flow increases the overhead in the broker. There is an upper limit to the number of nodes within a single flow. This limit is governed by system resources, particularly the stack size.

For more information about stack sizes, see “System considerations for message flow development” on page 75.

How the message flow routes and processes messages

In some situations, you might find that the built-in nodes, and perhaps other nodes that are available in your system, provide more than one way of providing the same function. Try to choose the simplest configuration. For example, if you want to define some specific processing based on the value of a field in each message, you might design a message flow that has a sequence of Filter nodes to handle each case. If appropriate, you can group messages through the Filter node to reduce the number that each message type has to pass through before being processed.

For example, you might have a message flow that handles eight different messages (Invoice, Dispatch Note, and so on). You can include a Filter node to identify each type of message and route it according to its type. You can optimize the performance of this technique by testing for the most frequent message types in the earliest Filter nodes. However, the eighth message type must always pass through eight Filter nodes.

If you can group the message types (for example, if the message type is numeric, and you can test for all types greater than four and not greater

than four), you can reduce the number of Filter nodes required. The first Filter node tests for greater than four, and passes the message on to two further Filter nodes (attached to the true and false terminals) that test for less than two and less than six respectively. An additional four Filter nodes can then test for one or two, three or four, and so on. Although the actual number of Filter nodes required is the same, the number of nodes that each message passes through is reduced.

You might find that using a RouteToLabel node with a set of Label nodes provides a better alternative to a sequence of Filter nodes. Each message passes through a smaller number of nodes, improving throughput. However, you must also consider using a RouteToLabel node means using a Compute node: the overhead of this node might outweigh the advantages. If you are dealing with a limited number of message types, a small number of Filter nodes is more efficient.

The Airline Reservations sample demonstrates how you can use the RouteToLabel and Label nodes instead of using multiple Filter nodes in the XML_PassengerQuery message flow. The Message Routing sample demonstrates how you can store routing information in a database table in an in-memory cache in the message flow.

If your message flow includes loops

Avoid loops of repeating nodes; these can be very inefficient and can cause performance and stack problems. You might find that a Compute node with multiple PROPAGATE statements avoids the need to loop round a series of nodes.

The efficiency of the ESQL

Check all the ESQL code that you have created for your message flow nodes. As you develop and test a node, you might maintain statements that are not required when you have finalized your message processing. You might also find that something you have coded as two statements can be coded as one. Taking the time to review and check your ESQL code might provide simplification and performance improvements.

If you have imported message flows from a previous release, check your ESQL statements against the ESQL available in Version 5.0 to see if you can use new functions or statements to improve its efficiency.

The use of persistent and transactional messages

Persistent messages are saved to disk during message flow processing. This is avoided if you can specify that messages either on input, output, or both, are non-persistent. If your message flow is handling only non-persistent messages, check the configuration of the nodes and the message flow itself; if your messages are non-persistent, transactional support might be unnecessary. The default configuration of some nodes enforces transactionality; if you update these properties and redeploy the message flow, response times might improve.

Message size

A larger message takes longer to process. If you can split large messages into smaller chunks of information, you might be able to improve the speed at which they are handled by the message flow. The Large Messaging sample demonstrates how to minimise the virtual memory requirements for the message flow to improve a message flow's performance when processing potentially large messages.

Message format

Although WebSphere Message Broker supports multiple message formats,

and provides facilities that you can use to transform from one format to another, this incurs overhead. Make sure that you do not perform any unnecessary conversions or transformations.

You can find more information on improving the performance of a message flow in this developerWorks article on message flow performance.

System considerations for message flow development

Default stack size

When a message flow thread executes, it requires storage to perform the instructions that are defined by the logic of its connected nodes. This storage comes from the execution group's heap and stack size. The default stack size allocated to a message flow thread depends on the platform used.

Each message flow thread is allocated 1MB of stack space.

Each message flow thread is allocated 8MB of stack space.

Each message flow thread is allocated 1MB of stack space.

Each message flow thread is allocated 512 KB of downward stack space and 50 KB of upward stack space.

In a message flow, a node typically uses about 2KB of the stack space. A typical message flow can therefore include roughly 250 nodes on z/OS, 500 nodes on UNIX platforms and 500 nodes on Windows. This amount can be higher or lower depending on the type of nodes used and the processing they perform.

Increasing the stack size on Windows and UNIX platforms

You can increase the stack size by setting the `MQSI_THREAD_STACK_SIZE` environment variable to an appropriate value. When you restart brokers running on the system they will use the new value.

The value of `MQSI_THREAD_STACK_SIZE` that you set is used for every thread that is created within a DataFlowEngine process. If the execution group has a large number of message flows assigned to it, and you set a large value for `MQSI_THREAD_STACK_SIZE`, the DataFlowEngine process will therefore need a large amount of storage for the stack.

Increasing the stack size on z/OS

Integrator components on z/OS are compiled using the new XPLINKage (extra performance linkage), which adds optimization to the runtime code. However, if the initial stack size is not large enough, then stack extents will be used. 128KB is used in each extent. It is very important that a large enough downward stack size is chosen, because XPLINK performs badly when stack extents are used.

To determine suitable stack sizes, a component administrator for z/OS can use the LE (Language Environment[®]) Report Storage tool. To do this, a message flow must be tested using the RPTSTG option with the `_CEE_RUNOPTS` environment variable. This should be set in the component profile (BIPBPROF for a broker) during the development and test of message flows intended for production. For example:

```
export _CEE_RUNOPTS=XPLINK\ (ON\),RPTSTG(ON)
```

You can then override the default values for the stack sizes on z/OS by altering or adding the LE_CEE_RUNOPTS environment variable in the component profile.

When updating the component profile you must stop the component, make the necessary changes to the profile, submit BIPGEN to recreate the ENVFILE and restart the component.

For example, you can change the default values of 50K and 512K in the following line to suit your needs:

```
export _CEE_RUNOPTS=XPLINK(ON),THREADSTACK(ON,50K,15K,ANYWHERE,KEEP,512K,128K)
```

Using RPTSTG increases the time an application takes to run. You should therefore use it as an aid to only the development of message flows, and your final production environment. When you have determined the correct stack sizes needed you should remove this option from the _CEE_RUNOPTS environment variable.

Note: XPLINK stacks grow downward in virtual storage while the old standard linkage grows upward. To avoid a performance impact caused by switching between downward stack space and upward stack space during runtime, you should compile user-defined extensions using the XPLINK option where possible. If your message flow uses user-defined extensions that have been compiled with the standard linkage convention, you will need to determine and set a suitable value for the upward stack size.

Determining the correct stack size

In WebSphere Message Broker any processing that involves nested or recursive processing can cause extensive usage of the stack. For example, in the following situations you might need to increase the stack size:

- When a message flow is processing a message that contains a large number of repetitions or complex nesting.
- When a message flow is executing ESQL that recursively calls the same procedure or function, or when an operator, for example the concatenation operator, is used repeatedly in an ESQL statement.

Related tasks

“Optimizing message flow response times” on page 73

Related reference

“Message flows” on page 479

Creating destination lists

You can create a list of destinations that indicates where a message is sent.

You can include a Compute node in your message flow, and configure it to create a destination list within the LocalEnvironment subtree. You can then use the destination list in the following nodes:

- The MQOutput node, to put output messages to a specified list of queues
- The RouteToLabel node, to pass messages to Label nodes

Refer to the Airline sample to see how this technique is used.

For more information about destination list contents, and example procedures for setting values for each of these scenarios, see “Accessing the LocalEnvironment tree” on page 192.

You might find it useful to create the contents of the destination list from an external database accessed by the Compute node. You can then update the destinations without having to update and redeploy the message flow.

The use of the destination list to define which applications receive the output messages is in contrast to the publish/subscribe application model, in which the recipients of the publications are those subscribers currently registered with the broker. The processing completed by the message flow does not have any effect on the current list of subscribers.

Using WebSphere MQ cluster queues for input and output

When you design the WebSphere MQ network that underlies your WebSphere Message Broker broker domain, consider whether to use clusters.

The use of queue manager clusters brings the following significant benefits:

1. Reduced system administration
Clusters need fewer definitions to establish a network; you can set up and change your network more quickly and easily.
2. Increased availability and workload balancing
You can benefit by defining instances of the same queue to more than one queue manager, thus distributing the workload through the cluster.

If you use clusters with WebSphere Message Broker, consider the following:

For SYSTEM.BROKER queues:

The SYSTEM.BROKER queues are defined for you when you create WebSphere Message Broker components, and are not defined as cluster queues. Do not change this attribute.

For broker, Configuration Manager, and User Name Server connectivity:

If you define the queue managers that support your brokers, the Configuration Manager, and the User Name Server to a cluster, you can benefit from the simplified administration provided by WebSphere MQ clusters. You might find this particularly relevant for the brokers in a collective, which must all have WebSphere MQ interconnections.

For message flow input queues:

If you define an input queue as a cluster queue, consider the implications for the order of messages or the segments of a segmented message. The implications are the same as they are for any WebSphere MQ cluster queue. In particular, the application must ensure that, if it is sending segmented messages, all segments are processed by the same target queue, and therefore by the same instance of the message flow at the same broker.

For message flow output queues:

- WebSphere Message Broker always specifies MQOO_BIND_AS_Q_DEF when it opens a queue for output. If you expect segmented messages to be put to an output queue, or want a series of messages to be handled by the same process, you must specify DEFBIND(OPEN) when you define that queue. This ensures that all segments of a single message, or

all messages within a sequence, are put to the same target queue and are processed by the same instance of the receiving application.

- If you create your own output nodes, specify MQOO_BIND_AS_Q_DEF when you open the output queue, and DEFBIND(OPEN) when you define the queue, if you need to guarantee message order, or to ensure a single target for segmented messages.

For publish/subscribe:

- If the target queue for a publication is a cluster queue, you must deploy the publish/subscribe message flow to all the brokers on queue managers in the cluster. However, the cluster does not provide any of the failover function to the broker domain topology and function. If a broker to which a message is published, or a subscriber registers, is unavailable, the distribution of the publication or registration is not taken over by another broker.
- When a client registers a subscription with a broker that is running on a queue manager that is a member of a cluster, the broker forwards a proxy registration to its neighbors within the broker domain; the registration details are not advertised to other members of the cluster.
- A client might choose to become a clustered subscriber, so that its subscriber queue is one of a set of clustered queues that receive any given publication. In this case, when registering a subscription, use the name of an "imaginary" queue manager that is associated with the cluster; this is not the queue manager to which the publication will be sent, but an alias for the broker to use. As an administrative activity, a blank queue manager alias definition is made for this queue manager on the broker that satisfies this subscription for all clustered subscribers. When the broker publishes to a subscriber queue that names this queue manager, resolution of the queue manager name results in the publication being sent to any queue manager that hosts the subscriber cluster queue, and only one clustered subscriber receives the publication. For example, if the clustered subscriber queue was SUBS_QUEUE and the "imaginary" subscriber queue manager was CLUSTER_QM, the broker definition would be:

```
DEFINE QREMOTE(CLUSTER_QM) RQMNAME(' ') RNAME(' ')
```

This sends broker publications for SUBS_QUEUE on CLUSTER_QM to one instance of the cluster queue named SUBS_QUEUE anywhere in the cluster.

To understand more about clusters, and the implications of using cluster queues, see the *WebSphere MQ Queue Manager Clusters* book.

Using WebSphere MQ shared queues for input and output (z/OS)

On z/OS systems you can define WebSphere MQ shared queues as input and output queues for message flows.

Use the WebSphere MQ for z/OS product facilities to define these queues and specify that they are shared.

For more information about configuring on z/OS, refer to the *WebSphere MQ for z/OS Concepts and Planning Guide*.

Using shared queues helps to provide failover support between different images running WebSphere Message Broker on a sysplex.

You cannot use shared queues for broker or User Name Server component queues such as SYSTEM.BROKER.CONTROL.QUEUE.

Shared queues are available only on z/OS.

Validating messages

The broker provides validation based on the message dictionaries for predefined messages. Validation, therefore, applies only to messages that you have modeled and defined to the MRM domain.

The broker does not provide any validation for self-defining messages. You can not validate messages directly against an XML DTD or XML Schema. Instead you should create the equivalent model in the MRM domain by importing your XML DTD or XML Schema using the Message Brokers Toolkit

Message flows are designed to transform and route messages that conform to certain rules. By default, the MRM parser performs some validity checking on a message, but only to ensure the integrity of the parsing operation. However, you can validate a message more stringently against the message model contained in the message dictionary, by specifying validation options on certain nodes in your message flow.

You can use validation options to validate the following messages:

- Input messages that are received by an input node
- Output messages that are created, for example, by a Compute, Mapping, or JavaCompute node

These validation options can guarantee the validity of data entering and leaving the message flow. The options provide you with some degree of control over the validation performed to:

- Maintain a balance between performance and security requirements.
- Validate at different stages of message flow execution, such as on input of a message and before a message is output, or at any point in-between
- Cope with messages that your message model does not fully describe.

You can also specify what action to take when validation fails.

Message validation involves navigating a message tree and checking its validity. It is an extension of tree creation when the message is parsed, and an extension of bit stream creation when the output message is written.

Validation options are available on the following nodes:

Node type	Nodes with validation options
Input node	MQInput, SCADAInput, HTTPInput, JMSInput, TimeoutNotification
Output node	MQOutput, MQReply, SCADAOutput, HTTPReply, JMSOutput

Node type	Nodes with validation options
Other nodes	Compute, Mapping, JavaCompute, Validate, ResetContentDescriptor, MQGet, HTTPRequest,

Validation options can also be specified on the ESQL CREATE statement and ASBITSTREAM function.

To validate input messages received on an input node, you can specify validation properties on the input node. The input message is then validated as the message bit stream is parsed to form the message tree.

You can also use the Parse Timing property of the input node to control whether the entire message is to be parsed and validated at this time, or whether individual fields in the message are parsed and validated only when referenced.

To validate output messages created by a Compute node, a Mapping node, or a JavaCompute node, you either specify validation properties on the node itself, or you specify validation properties on the output node that sends the message. The validation takes place when the message bit stream is created from the message tree by the output node.

Alternatively, you can use a Validate node to validate a message tree at a particular place in your message flow, or you can use the ESQL ASBITSTREAM function within a Compute, Filter or Database node.

A limited amount of validation occurs by default if you leave the validation settings unaltered. At this default level, an exception is thrown if one of the following is true:

- There is a data mismatch, where, for example, the parser cannot interpret the data provided for the field type specified
- The order of elements in the output message does not match the order in the logical message tree (CWF and TDS fixed length models only)

Additionally, the MRM parser by default will perform limited remedial action under the following circumstances:

1. Extraneous fields are discarded on output for fixed formats (CWF and TDS fixed length models only)
2. If mandatory content is missing, defaults are supplied, if available, on output for fixed formats (CWF and TDS fixed length models only)
3. If an element's data type in the tree does not match that specified in the dictionary the data type is converted on output to match the dictionary definition if possible, for all formats.

However, using validation options you can request more thorough validation of messages. For example, you might want to validate one or more of the following conditions and throw an exception, or log the errors:

- The whole message at the start of the message flow
- That complex elements have the correct *Composition* and *Content Validation*
- That all data fields contain the correct type of data
- That data fields conform to the value constraints in the message model
- That all mandatory fields are present in the message

- That only the expected fields are present in the message
- That message elements are in the correct order

The sample illustrates some of these validation options.

When using validation options, it is important to understand the following behaviour:

- The Parse Timing property, which controls whether 'on demand' parsing (sometimes called partial parsing) takes place, has an effect on the timing of the validation of input messages, including message headers.
For more information on Parse Timing, see "Validation properties for messages in the MRM domain" on page 703.
- If a message tree is passed to an output node, then by default the output node inherits the validation options in force for the message tree. You can override these options by specifying a new set of validation options on the output node.
- If a message tree is passed as input to a Compute, Mapping, or JavaCompute node, any new output message trees created by the Compute, Mapping, or JavaCompute node have the validation options specified by the node itself (even if the whole message is copied). You can override this behaviour and specify that messages created by the node inherit the validation options of the input message tree.
- When the bitstream is written and validation options are applied, the entire message is validated. It is possible that the message tree contains an unresolved type (for example, if a Compute node copied an unresolved type from an input message to an output message without resolving it). If such a type is encountered, a validation error occurs because it is not possible to validate the type. To prevent this, ensure that all unresolved types are resolved before they are copied to output messages.

For information about how you can control validation by using different properties, see "Validation properties for messages in the MRM domain" on page 703.

Viewing the logical message tree in trace output

To view the structure of the logical message tree at any point in the message flow, include a Trace node and write some or all of the message (including headers and all four message trees) to the trace output destination.

You might find this useful to check or record the content of a message before and after a node has made changes to it, or on its receipt by the input node. For example, if you include a Compute node that builds a destination list in LocalEnvironment, you might want a record of the structure that it has created as part of an audit trail, or you might just want to check that the Compute node is working as you expect it to.

1. Switch to the Broker Application Development perspective.
2. Open the message flow for which you want to view messages. This can be an existing message flow, or you can create a new message flow.
3. Include a Trace node wherever you want to view part or all of the message tree structure. There is no limit to the number of Trace nodes that you can include, however each node introduces some overhead to the message flow processing.

4. Set the Trace node properties to trace the message, or parts of the message, that you want to view. You specify the parts of the message using ESQL field references. Several examples are included below.
5. If you have added a Trace node to investigate a particular behavior of your message flow, and have now resolved your concerns or checked that the message flow is working correctly, remove the Trace node or nodes, and redeploy the message flow.

Assume that you have configured a message flow that receives an XML message on a WebSphere MQ queue in an MQInput node. The input message includes an MQRFH2 header. The message content is shown below:

```
<Trade type='buy'  
  Company='IBM'  
  Price='200.20'  
  Date='2000-01-01'  
  Quantity='1000'/>
```

You can include and configure a Trace node to produce output that shows one or more of the trees created from this message: the message body, Environment, LocalEnvironment, and Exception trees. If you choose to record the content of the message body, the Properties tree and the contents of all headers (in this example, at least an MQMD and an MQRFH2) are included. You specify what you want to be recorded when you set the Trace node property *Pattern*. You can use most of the correlation names to define this pattern (you cannot use those names that are specific to the Compute node).

Message body

If you want the Trace node to write the message body tree including Properties and all headers, set *Pattern* to `$Root`. If you want only the message data, set *Pattern* to `${Body}`.

The trace output generated for the message tree of the message shown above with *Pattern* set to `$Root` would look something like:

```

Root
  Properties
    CreationTime=GMTTIMESTAMP '1999-11-24 13:10:00'      (a GMT timestamp field)
  ... and other fields ...

  MQMD
    PutDate=DATE '19991124'                               (a date field)
    PutTime=GMTTIME '131000'                              (a GMTTIME field)
  ... and other fields ...

  MQRFH
    mcd
    msd='xml'                                             (a character string field)
  .. and other fields ...

  XML
    Trade
    type='buy'                                            (a character string field)

    Company='IBM'                                         (a character string field)

    Price='200'                                           (a character string field)

    Date='2000-01-01'                                     (a character string field)

    Quantity='1000'                                       (a character string field)

```

Environment

To trace any data in the environment tree, set *Pattern* to `${Environment}`. This produces output similar to the following:

```

(0x1000000)Environment = (
  (0x1000000)Variables = (
    (0x1000000)MyVariable1 = (
      (0x2000000) = '3'
    )
    (0x1000000)MyVariable2 = (
      (0x2000000) = 'Hello'
    )
  )
)

```

To trace particular variables in the variables folder of the environment tree, you can use a more specific pattern, for example `${Environment.Variables.MyVariable1}`. This returns the value only (for example, it returns just the value 3).

LocalEnvironment

To trace data in the LocalEnvironment tree, set *Pattern* to `${LocalEnvironment}`. The output you get is similar to the following example, which shows that a destination list has been created within the LocalEnvironment tree:

```

(0x1000000)Destination = (
  (0x1000000)MQ = (
    (0x1000000)DestinationData = (
      (0x3000000)queueName = 'MQOUT'
    )
  )
  (0x1000000)MQDestinationList = (
    (0x1000000)DestinationData = (
      (0x3000000)queueName = 'OLDMQOUT'
    )
  )
  (0x1000000)RouterList = (
    (0x1000000)DestinationData = (
      (0x3000000)labelName = 'continue'
    )
    (0x1000000)DestinationData = (
      (0x3000000)labelName = 'custdetails'
    )
    (0x1000000)DestinationData = (
      (0x3000000)labelName = 'trade'
    )
  )
)
)

```

Another example, shown below, includes a `WrittenDestination` folder. This represents a trace that has been written by a `Trace` node included after an `MQOutput` node, where the out terminal of the `MQOutput` node is connected to a sequence of nodes including the `Trace` node. When an out terminal is connected, the `LocalEnvironment` is augmented with information about the action that the output node has performed.

```

(0x1000000)Destination = (
  (0x1000000)MQ = (
    (0x1000000)DestinationData = (
      (0x3000000)queueName = 'MQOUT'
    )
  )
  (0x1000000)WrittenDestination = (
    (0x1000000)MQ = (
      (0x1000000)DestinationData = (
        (0x3000000)queueName = 'MQOUT'
        (0x3000000)queueManagerName = 'MQSI_SAMPLE_QM'
        (0x3000000)replyIdentifier = X'414d51204d5153495f53414d504c455f1f442f3b12600100'
        (0x3000000)msgId = X'414d51204d5153495f53414d504c455f1f442f3b12600100'
        (0x3000000)correlId = X'0000000000000000000000000000000000000000000000000000000000000000'
      )
    )
  )
)
)
)

```

ExceptionList

To trace data in the exception list, set *Pattern* to `${ExceptionList}`.

You can also view message structure within the message flow, and other information, when you use the flow debugger.

Accessing databases from message flows

You can create and configure message flows to access user databases.

Before you start:

To complete this task, you must have completed the following tasks:

- “Creating a message flow” on page 126
- Creating the databases

You can access additional information in a database to enhance or influence the operation of the message flow, and modify the contents of a database by adding new information or removing or replacing existing information.

You can access a database from a message flow from the following nodes:

- Compute
- Database
- DataInsert
- DataDelete
- DataUpdate
- Filter
- Mapping
- Warehouse

For more details of these nodes, and how to configure them in message flows, see “Built-in nodes” on page 482.

If you want the actions that the message flow takes against the database to be coordinated with other actions, configure the message flow to support coordinated transactions. For information on how to do this, see “Configuring coordinated message flows” on page 93.

To access a database from a message flow:

1. Identify the database that you want to access. This can be an existing database or a new one created for this purpose.

If you want to create a new database, follow the instructions given in Creating the databases. These describe how to create a DB2 database for a broker, but are equally applicable for user databases.

If you want to use a database other than DB2, refer to the database product documentation for details on how to do this.

Supported databases defines the database products that are supported by WebSphere Message Broker.

2. Create an ODBC data source to the database if one does not already exist. Follow the instructions given in Configuring access to databases. These describe how to create a data source for a broker database, but are equally applicable for user databases.
3. Authorize the broker to access the database.

Access to a user database from within a message flow is controlled by user ID and password.

On z/OS, you can specify these values:

- a. When you create the broker.

The broker started task ID is used to access user databases, irrespective of the user ID and password specified on the mqsicreatebroker command in the BIPCRBK JCL in the customization data set <hlq>.SBIPPROC.

- b. After you have created the broker.

Use the BIPSDBP JCL in the customization data set <hlq>.SBIPPROC to customize the mqsisetdbparms command to specify a user ID and password pair for a specific database. This changes the defaults that were set when you created the broker (described above)

You can create a user ID and password pair for any database (identified by DSN) that is accessed by a message flow. You can therefore control access to a database at an individual level if you choose. This includes databases that you have created and configured on distributed systems that are accessed by z/OS DB2 remote database access.

On distributed systems, you can specify these values:

- a. When you create the broker.

The `mqsicreatebroker` command has two parameters `-u DataSourceUserid` and `-p DataSourcePassword` that you can use to identify the user ID that the broker uses to access its own database. If you specify these parameters, they are used as the default access control parameters for user databases that are accessed by message flows.

If you do not specify `DataSourceUserid` and `DataSourcePassword`, the broker uses the values specified for the parameters `-i ServiceUserID` and `-a ServicePassword` (which identify the user under which the broker runs) as the default values.

- b. After you have created the broker.

Use the `mqsisetdbparms` command to specify a user ID and password pair. This changes the defaults that were set when you created the broker (described above).

You can create a user ID and password pair for any database (identified by DSN) that is accessed by a message flow. You can therefore control access to a database at an individual level if you choose. This includes databases that you have created and configured on z/OS that are accessed by brokers on distributed systems.

If the user that created a table in a database is not the user that the broker is using to access the database, you must specify the user ID that created the database as the schema name in relevant ESQL statements, unless you have set up an alias or synonym.

Note: If you access a database from a message flow using a Compute, Database or Filter node, import the database SQL into the message flow project or reference another project to which the database SQL has been imported. Use the Data perspective to create a connection to the appropriate database, then import it into the message flow project.

The Message Routing sample, the Data Warehouse sample, the Error Handler sample, and the Airline Reservations sample access databases from message flows. The Message Routing sample and Data Warehouse sample use Compute nodes to access the database, the Error Handler uses Database nodes to access the database, and the Airline Reservations sample uses both Compute and Database nodes.

Accessing databases from ESQL

You can create and configure ESQL in message flows to access user databases.

Before you start:

To complete this task, you must have completed the following tasks:

- “Creating a message flow” on page 126
- Creating the databases

You can use a number of ESQL statements and functions to access databases:

INSERT statement

The INSERT statement adds a row to a database table

UPDATE statement

The UPDATE statement changes one or more values stored in zero or more rows

DELETE FROM statement

The DELETE FROM statement removes zero or more rows

SELECT function

The SELECT function retrieves data from a table

CALL statement

The CALL statement invokes a stored procedure

PASSTHRU statement

The PASSTHRU statement can be used to invoke administrative operations, such as creating a table

PASSTHRU function

The PASSTHRU function can be used to invoke complex selects

You can access user databases from Compute, Database, and Filter nodes. There is no difference between the database access capabilities of these nodes, but the following restrictions apply.

- Any node which uses any of the above database constructs must have its "data source" attribute set with the name (i.e. ODBC DSN) of a database and that database must be accessible, operational and allow the broker to connect to it
- All databases accessed from one node must present the same ODBC functionality. This requirement is certainly satisfied if they are of the same type (e.g. DB2, Oracle) and at the same level (e.g. release 8.1 CSD3)
- All tables referred to in a single SELECT function's FROM clause must lie in the same database

To access databases, you must ensure that suitable ODBC data sources have been created on the system on which the broker is running. If you have used the `mqssetdbparms` command to set a user ID and password for a particular database, the broker uses these values to connect to the database. If you have not set values for a particular database, the broker uses the default database user ID and password that you supplied on the `mqscreatebroker` command, or the user ID and password details that you specified if you have modified them using the `mqschangebroker` command.

On z/OS systems, the broker uses the broker started-task ID to connect to the database. You must also ensure that the database user IDs have sufficient privileges to perform the operations your flow requires. If they do not have the required privileges then errors will occur at runtime.

For a description of database transactional issues, see "The Transactional model" on page 88.

You are recommended to set the "throw exception on database error" and "treat warnings as errors" attributes to 'yes' and the "transaction" attribute to "automatic" as this gives the maximum flexibility. You then use the COMMIT and ROLLBACK statements for transaction control and handlers for dealing with errors.

The Transactional model

A message flow can be considered to consist of the following constituent parts:

- An input queue
- The message flow
- One or more database tables
- One or more output queues

The sequence of events when a flow processes a message can be considered to be:

1. A message is taken from the input queue
2. Data is read from or written to the database tables and queues
3. The system quiesces and awaits the next input message

Note that this sequence of events makes no distinction between the accessing of tables and the writing of output messages. Although a flow is usually thought of as producing some sort of output message, there is no real distinction between this and updating a database table. In both cases there is some change to the state of the data in the system.

Consider the following diagram:

```
====x=====x===x=====x=====x====x=====
      1         2 3         4         5         6
```

The line represents the data in the system as time passes. At time 1 a message arrives and is taken from the input queue. At times 2, 3, 4 and 5, database tables are updated or queues are written to. These changes of state are indicated by the x symbol. Finally at time 6 the output messages depart and the system quiesces. Between these events there is no change to the state of the data and this is indicated by the = symbol.

Now let us consider the effect of failures on the system. If a failure (e.g. the plug is pulled on the machine on which the broker is running), the changes to the state of tables and queues made prior to the failure will have taken place but those that should have occurred after will not take place at all. It is quite likely that such a situation is unacceptable (e.g. a payment from my current account to my mortgage account has been taken from the current account table but not added to the mortgage account table).

Transactions

To avoid this problem, the broker, queuing systems and databases have a transactional model. The basis of these is that, as processing proceeds, additional data is stored which allows the original state to be restored in the event of a failure. Exactly what form this data takes and exactly how this is achieved doesn't concern us. The following diagram illustrates the state of this extra data:

```
====x=====x===x=====x=====x====x=====
      1         2 3         4         5         6
```

The line represents the extra data in the system as time passes. At time 1 a message arrives and is taken from the input queue. Before this there is no extra data in the system and this is indicated by the - symbol. After this time, the state represents the fact that a message has been taken from the queue so that, if need be, it can be put back. At times 2, 3, 4 and 5 database tables are updated or queues are written to. Again the state of the extra data changes so that the changes to tables and queues can be undone if need be. Finally at time 6 the output messages depart, the system quiesces and there is no extra data in the system once more.

Between these events there is no change to the state of the extra data and this is indicated by the symbol =. If a failure occurs at any time between 1 and 6 the extra data is used to restore the original state of the system's data so that, in effect, none of the output queues have been written to, none of the tables have been changed and the input message hasn't been taken from the input queue. If no failure occurs the changes become permanent at time 6, that is, an undo operation following a subsequent failure will not undo them.

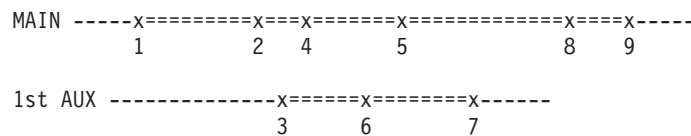
This mode of operation is known as coordinated transaction mode. The successful completion of a transaction is known as its commit. Unsuccessful completion is known as rollback. (See section XYZ for a long and complicated description of how to achieve it, be aware that it's not the default and that not all databases or queuing systems support it.)

Uncoordinated Auxiliary Transactions

The key feature of the coordinated transaction mode of operation is that either all the changes to queues and tables associated with one input message are made or none are made regardless of where or when failures occur and this is its key advantage. It does not suit all situations however. Two simple examples of such situations are:

- We want to make a list (an audit log) of all attempts at processing. The writes to this list need to be committed even when updates to the main tables and queues are rolled back
- We want to send acknowledgment or not-acknowledgment message back to the originator of the messages we are processing according to whether the message processing succeeds or fails. These messages need to be actually sent (i.e. committed) even when the updates to the main tables and queues are rolled back

To satisfy such requirements WebSphere Message Broker allows writes to queues and tables to be take place in a separate transaction. This is illustrated in the following diagram:



The upper line represents the main transaction as before. Transaction is a rather abstract term but what it equates to in the real world is the extra data needed to restore the original state should the need arise. The lower line represents an auxiliary transaction. At time 3 an update to a table (or queue) is made. Another is made at time 6. At time 7 the flow decides that all the changes that need to be made under the auxiliary transaction are complete and commits the changes.

If the flow were to fail before time 7, the state of the system would be unchanged since both transactions would be rolled back. If failure occurs after time 7 but before time 9, the auxiliary transaction would be committed (it already has been) but the main one would be rolled back. If a failure hasn't occurred by time 9, there is no failure and both transactions are committed.

Database Auxiliary Transactions

You are not restricted to a single auxiliary transaction and you are not restricted to only committing them. A number of updates can be made to database tables and

these can be either committed or rolled back. You can then make some more changes to the same database tables, or to different tables, and then either commit or rollback these changes.

Each database that you use has its own auxiliary transaction and thus, if the flow updates tables belonging to different database instances (i.e. different data source names) there will be an auxiliary transaction per database. These may be (indeed must be) committed or rolled back individually. Any updates which have not been either committed or rolled back when the operation completes (time 9) will be automatically committed or rollback by the broker according to whether the processing succeeded or failed (i.e. whether an exception reaches the input node or not).

Note that some databases types, such as DB2 on AIX, do not allow both coordinated and uncoordinated transactions in the same database instance. In these cases, separate database instances must be created. You should configure one database instance for coordinated transactions and the other for uncoordinated transactions.

Auxiliary database transactions are committed or rolled back using the ESQL COMMIT and ROLLBACK statements. Operations outside the main transaction are obtained by specifying UNCOORDINATED on the individual database statements used, such as the INSERT and UPDATE statements.

Queue Auxiliary Transactions

Not all queuing systems have the capability of databases described above. In the case of MQ, each individual uncoordinated read or write to a queue has an implied commit. You cannot thus put two messages and then decide to commit them both or to roll them both back. This is a limitation of the underlying system which WebSphere Message Broker cannot hide. The COMMIT and ROLLBACK statements thus only operate on databases.

Nodes

The above description talks only about flows and does not mention nodes. The way a flow is divided into nodes is, in principle, of no relevance to a discussion about transactions. Any number of nodes may make updates to the main and any number of auxiliary transactions in any way they see fit without restriction. What you do matters but how you do it does not. In practice however this is only true for operations on databases simply because other data sources (VSAM, Queues) have more limited capabilities.

Entirely Uncoordinated Transactions

A special case worth mentioning is when all database updates are done within auxiliary transactions. In this case, the "Coordinated Transaction" attribute of the message flow can be set to "no". This makes all table references outside the main transaction saving the trouble of having to specify it on each database operation. It may also make flows faster.

Note: It is still possible in this case for writes to MQ queues to be coordinated with the getting of the input message from an MQ queue.

Message flow aggregation

Aggregation is the generation and fan-out of related requests derived from a single input message and the fan-in of the corresponding replies to produce a single aggregated reply message.

The initial request received by the message flow, representing a collection of related request items, is split into the appropriate number of individual requests to satisfy the subtasks of the initial request. This process is known as fan-out and is provided by a message flow that includes aggregation nodes.

Replies from the subtasks are combined and merged into a single reply that is returned to the original requester (or another target application) to indicate the completion of the processing. This process is known as fan-in, and is also provided by a message flow that includes aggregation nodes.

You can initiate aggregation by sending a message to a message flow from a client application that is connected to the broker across any supported protocol; the aggregated response can also be sent to a client application across all these protocols. The messages issued by the fan-out message flow, and the responses received by the fan-in message flow, must be request/reply messages. You are therefore limited to client applications that connect using the WebSphere MQ Enterprise Transport (sending and receiving messages to and from the MQInput and MQOutput nodes) or to clients that use another protocol supported by user-defined input and output nodes that conform to the request/reply communication model.

WebSphere Message Broker provides three message flow nodes that support aggregation:

- AggregateControl
- AggregateRequest
- AggregateReply

When you include these nodes in your message flows, the multiple fan-out requests are issued in parallel from within a message flow. This is in contrast to the standard operation of the message flow in which each node performs its processing in sequence.

You can also use these nodes to issue requests to applications outside the broker environment; messages can be sent asynchronously to external applications or services, the responses retrieved from those applications, and the responses combined to provide a single response to the original request message.

These nodes also provide an opportunity for an improvement in response time, because slow requests can be performed in parallel and do not have to follow each other sequentially. If the subtasks can be processed independently, and do not have to be handled as part of a single unit of work, they can be processed by separate message flows.

You can design and configure a message flow that provides a similar function without using the aggregate nodes, by issuing the subtask requests to another application (for example, using the HTTPRequest node) and recording the results of each in the LocalEnvironment. After each subtask has completed, you can merge the results from the LocalEnvironment in a Compute node, and create the combined response message for propagating to the target application. If you do this, all the subtasks are performed sequentially, and do not provide the performance benefits of parallel operation that you can achieve using the aggregate nodes.

Examples of aggregation flows using the aggregate nodes are provided in the Aggregation sample and Airline Reservations sample. The Aggregation sample demonstrates a simple four-way aggregation, and the Airline Reservations sample

simulates requests related to an airline reservation service, and illustrates the techniques associated with aggregation flows.

Data conversion

Data conversion is the process by which data is transformed from the format recognized by one operating system into that recognized by a second operating system with different characteristics such as numeric order.

If you are using a network of systems that use different methods for storing numeric values, or you need to communicate between users who view data in different code pages, you must consider how to implement data conversion.

Numeric order

For numeric and encoding aspects, consider:

- Big Endian versus Little Endian
- Encoding values in WebSphere MQ (the Encoding field in the MQMD)
Encoding values are system specific. For example, Windows usually has an encoding of 546, hexadecimal value X'00000222'. The three final hexadecimal digits identify:

1. The float number format

This value can be 1 (IEEE format byte order normal), 2 (IEEE format byte order reversed), or 3 (zSeries format byte order normal).

2. The packed decimal number format

This value can be 1 (byte order normal) or 2 (byte order reversed).

3. The hexadecimal number format

This value can be 1 (byte order normal) or 2 (byte order reversed).

The bit order within a byte is never reversed. Byte order normal means that the least significant digit occupies the highest address.

Systems that process numbers in normal byte order are Big Endian (z/Series, iSeries, Linux, and UNIX). Systems that process numbers in reversed byte order are Little Endian (mainly PCs).

For further details about numeric order, see Appendix D, Machine Encodings, in the *WebSphere MQ Application Programming Reference*.

Code page conversions

Code page conversion might be required for any of the following reasons:

- ASCII versus EBCDIC
- National languages
- Operating system specific code pages

For more information about code page support in WebSphere MQ, see the *WebSphere MQ Application Programming Reference* book.

When you use WebSphere Message Broker, you can use the data conversion facilities of WebSphere MQ, WebSphere Message Broker, or both.

WebSphere MQ facilities

Headers and message body are converted according to the MQMD values, and other header format names. You might have to set up data conversion exits to convert the body of your messages.

When you use WebSphere MQ facilities, the whole message is converted to the specified encoding and CCSID, according to the setting of the format in the WebSphere MQ header.

For more detail about data conversion using WebSphere MQ facilities, see Appendix F, Data Conversion, in the *WebSphere MQ Application Programming Reference*.

WebSphere Message Broker facilities

You can model your messages in the MRM through the workbench. Predefined elements of the messages are converted according to their type and physical layer characteristics. For further details, see *Configuring physical properties*. You can also use self-defining messages. You can then use the Compute node to configure encoding and CCSIDs. You do not need WebSphere MQ data conversion exits.

- String data is converted according to the CCSID setting.
- Decimal integer and float extended decimal types are converted according to the CCSID setting.
- Decimal integer and float (other physical data types) are converted according to the Encoding setting.
- Binary and boolean data is not converted.

WebSphere Message Broker can also convert those WebSphere MQ headers for which parsers are provided (listed in “Parsers” on page 26).

When you use WebSphere Message Broker facilities, the whole message is not converted to the specified encoding and CCSID: you can specify a different encoding, or CCSID, or both, in each header to perform a different conversion for the following part of the message. The encoding and CCSID in the last header defines the values for the message body.

Configuring coordinated message flows

If you want the actions of a message flow to be coordinated (that is, it must complete all processing successfully, or complete none), ensure your configuration supports this.

Review the following tasks, and complete the actions that are appropriate for your requirements:

- “Configuring message flow transactions”
- “Configuring databases for coordinated message flows” on page 96
- “Configuring WebSphere MQ and RRS for coordinated message flows” on page 106

The Error Handler sample demonstrates the use of globally-coordinated transactions, and the differences in the message flow when database updates are coordinated (the main flow) and when they are not (the error flow).

Configuring message flow transactions

If you want to coordinate the message flow processing with other resources, you must configure properties both of the nodes within the message flow and of the message flow itself.

Before you start:

To complete this task, you must have completed the following task:

- “Creating a message flow” on page 126

A coordinated message flow executes within a single transaction, which is started when a message is received by an input node, and can be committed or rolled back when all processing has completed. You can also control how database errors are handled by the node that interacts with the database.

To configure the message flow and the nodes:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with, or create a new message flow.
3. Set the *Transaction* property for the following nodes if they appear in this message flow:
 - Compute
 - Database
 - DataDelete
 - DataInsert
 - DataUpdate
 - Filter
 - Mapping
 - Warehouse

You can set the *Transaction* property to the following values:

Automatic

Any updates, deletions, and additions performed by the node are committed or rolled back when the message flow processing completes. If the message flow completes successfully, all changes are committed. If the message flow does not complete successfully, all changes are rolled back.

If you want all processing by the message flow to be coordinated, you must select this value.

Commit The action taken depends on the system to which the message flow has been deployed:

- On distributed systems, any work that has been done to this data source in this message flow to date, including any actions taken in this node, is committed regardless of the subsequent success or failure of the message flow.

Note: On platforms other than z/OS, individual relational databases may or may not support this mode of operation.

- On z/OS, actions taken in this node *only* are committed regardless of the subsequent success or failure of the message flow. Any actions taken before this node under automatic transactionality are not committed, but remain within a unit of work and might either be committed or rolled back depending on the success of the message flow.

If you want to mix nodes with Automatic and Commit transactionality in the same message flow, where the nodes operate on the same external database, you must use separate ODBC connections: one for the nodes that are not to commit until the completion of the message flow, and one for the nodes that are to commit immediately. If you do not, the nodes that commit immediately will also commit all operations carried out by preceding Automatic nodes.

Note: On platforms other than z/OS, individual relational databases may or may not support this mode of operation.

If you define more than one ODBC connection you might get database locking problems. In particular, if a node with Automatic transactionality carries out an operation, such as an INSERT or an UPDATE, that causes a database object (such as a table) to be locked, and a subsequent node tries to access that database object using a different ODBC connection, an infinite lock (deadlock) occurs.

The second node waits for the lock acquired by the first to be released, but the first node will not commit its operations and release its lock until the message flow completes; this will never happen because the second node is waiting for the first node's database lock to be released.

Such a situation cannot be detected by any DBMS automatic deadlock-avoidance routines because the two operations are interfering with each other indirectly, using the broker.

There are two ways to avoid this sort of locking problem:

- Design your message flow so that uncommitted (automatic) operations do not lock database objects that subsequent operations using a different ODBC connection need to access.
- Configure your database's lock timeout parameter so that an attempt to acquire a lock fails after a specified length of time. If a database operation fails due to a lock timeout, an exception is thrown that the broker handles in the normal way.

For information concerning which database objects are locked by particular operations, and how to configure your database's lock timeout parameter, consult your database product documentation.

4. Set the *Transaction Mode* property for the following nodes, if they appear in this message flow:
 - MQInput
 - MQOutput
 - MQReply
 - SCADAInput
 - JMSInput node
 - JMSOutput node

The table below provides a summary of the actions taken in response to specific property settings for the input and output nodes.

Message persistence ^a	Input node Transaction Mode	MQOutput or MQReply node Transaction Mode	Message flow is globally coordinated?
Yes	Yes	Automatic	Yes
No	Yes	Automatic	Yes
Yes	No	Automatic	No
No	No	Automatic	No
Yes	Automatic	Automatic	Yes
No	Automatic	Automatic	No
Any ^b	Any ^b	Yes	Yes
Any ^b	Any ^b	No	No

Notes:

- a. Persistence is relevant only for messages received across the WebSphere MQ Enterprise Transport, WebSphere MQ Mobile Transport, and WebSphere MQ Telemetry Transport protocols.

- b. The MQOutput or MQReply node property setting overrides the value set here.
- c. The Transaction Mode settings of the JMSInput and JMSOutput nodes are set differently to the above table. See “JMSInput node” on page 546 and “JMSOutput node” on page 558 for information.

The default on each input node is Yes, which means that the incoming messages are processed under syncpoint. In addition, messages sent to the output node are delivered under syncpoint. You can change this behavior if the output node is an MQOutput or MQReply node, both of which have a *Transaction Mode* property.

If you set the *Transaction Mode* on an input node to Automatic, the incoming messages are processed under syncpoint only if they are defined as persistent. Messages sent to the MQOutput node are delivered under syncpoint unless you explicitly change the *Transaction Mode* in the MQOutput node.

5. Set the *Treat warnings as errors* and *Throw exception on database error* for each node that accesses a database to indicate how you want that node to handle database warnings and errors. Whether you select these properties, and how you connect the failure terminals of the nodes, also affect the way in which database updates are committed or rolled back.
6. Switch to the Broker Administration perspective.
7. Add the message flow to a broker archive.
8. Select the Configure tab below the broker archive editor view and select the message flow. This displays the configurable properties for the message flow within the broker archive. Select the check box *coordinatedTransaction* to configure the message flow as globally coordinated.

On z/OS, transactions are always globally coordinated. The setting of the *coordinatedTransaction* property for a message flow is ignored. Coordination is always provided by RRS.

Configuring databases for coordinated message flows

If your message flow interacts with a database, and you want to coordinate the updates made to the database with other actions within the message flow, configure your broker to manage these updates.

Before you start:

To complete this task, you must have completed the following task:

- “Configuring message flow transactions” on page 93
1. Update the broker queue manager information by defining an XAResourceManager stanza for each of the databases that will participate in globally coordinated transactions when updated by message flows on that broker.
 - On Linux and UNIX, add an XAResourceManager: stanza to the broker queue manager’s initialization file qm.ini. The content of this stanza is database specific; see the instructions in the sections that follow.
 - On Windows, define the XAResourceManager using either the WebSphere MQ Explorer or WebSphere MQ Services depending on which version of WebSphere MQ you are using.

WebSphere MQ Version 6 or later

- a. Open the WebSphere MQ Explorer.

- b. Right click the queue manager name in the left pane and select **Properties....**
- c. Click **XA resource managers**.
- d. Click **Add...**
- e. Enter the values indicated in the following topics.
- f. Click **OK**.
- g. Click **Apply**.
- h. Click **OK**.

WebSphere MQ Version 5.3.1 or earlier

- a. Open WebSphere MQ Services.
 - b. Click the queue manager name in the left pane and select **Properties** from the menu.
 - c. In the Resources tab of the Properties dialog, enter the values indicated in the following topics.
 - On z/OS, you do not have to take any specific action, but RRS must be available.
2. If your message flows reference message dictionaries, or contain Publication nodes, you must also define an `XAResourceManager` stanza for the broker internal database using the same method.

The following topics explain how to coordinate message flows using different databases:

- “Configuring databases for coordinated message flows using DB2”
- “Configuring databases for coordinated message flows with Oracle” on page 99
- “Configuring databases for coordinated message flows with Sybase” on page 103

64-bit or 32-bit coordination:

The database drivers that are needed for XA coordination will depend on the version of WebSphere MQ that you will be using to perform transaction management. If you are going to use WebSphere MQ V5.3 then you can only perform XA coordination in 32-bit mode and so you must choose and configure the 32-bit drivers in both your `qm.ini` and `ODBCINI` files.

If you are going to use WebSphere MQ V6 then the XA coordination will involve 64-bit mode and you should choose and configure the 64-bit drivers in your `qm.ini` and `ODBCINI64` files. If any 32-bit execution groups are being used, then both 64-bit and 32-bit drivers must be configured.

Configuring databases for coordinated message flows using DB2:

If your message flow interacts with a DB2 database, and you want to coordinate the updates made to the database with other actions within the message flow, configure your broker to manage these updates.

Before you start:

To complete this task, you must have completed the following task:

- “Configuring databases for coordinated message flows” on page 96

If you are using DB2 Version 8 on Linux, UNIX, or Windows systems, DB2 Version 8.1 Fix Pack 5 is the minimum level that is supported and you must configure the following values.

1. Database configuration:

- You must set the transaction processor monitor name (TP_MON_NAME) to MQ on Windows. You must **not** set this parameter if you are using a 64-bit DB2 instance. The setting of this variable is optional on 32-bit instances of DB2 on Linux and UNIX.
- Ensure that you have adequate connection resources; either use all TCP/IP connections or enable extended shared memory.
- To enable shared memory support for DB2, take the following steps:

- Turn on extended shared memory in the DB2 server:

```
export EXTSHM=ON
db2set DB2ENVLIST=EXTSHM
db2stop
db2start
```

- Turn on shared memory support in the broker environment:

```
mqsistop broker
export EXTSHM=ON (in the profile of all broker environments)
mqsistart broker
```

2. Queue Manager configuration:

The toc (Thread of Control) specified in the XAOpenString determines the mode in which the Resource Manager (DB2) runs. The ThreadOfControl stanza specifies the mode in which the Transaction Manager (WebSphere MQ) runs. Specify explicit values to ensure that both products run in THREAD mode.

Using DB2 with only 32-bit execution groups:

If you want to use DB2 in coordinated transactions:

- If you use DB2 Version 8, use the supplied switch file; a file called db2swit is provided for each supported platform. These files are installed in the directory <install_dir>/sample/xatm.
- If you prefer to build your own files, follow the instructions in the section "DB2 configuration" in the *WebSphere MQ System Administration Guide*. You can also find details of how to configure XA resource definitions in this book.

Refer to the information provided for the version of WebSphere MQ that you have installed:

- For coordination by WebSphere MQ V5:

The following examples show what you must include in the XAResourceManager stanza in the qm.ini file on Linux and UNIX systems, and the equivalent information for Windows.

On Linux and UNIX (AIX, HP-UX, and Solaris):

```
XAResourceManager:
Name=DB2
SwitchFile=install_dir/sample/xatm/db2swit
XAOpenString=db=yourdatabase,uid=youruserid,pwd=yourpassword,toc=t
XACloseString=
ThreadOfControl=THREAD
```

On Windows, set the following values on the Resources page of the properties dialog for your WebSphere MQ queue manager. This example assumes that you have installed WebSphere Message Broker in the directory C:\WMQI:

```
SwitchFile: C:\wmqi\sample\xatm\db2swit.dll
XAOpenString=db=yourdatabase,uid=youruserid,pwd=yourpassword,toc=t
ThreadOfControl=THREAD
```

- For coordination by WebSphere MQ V6:

The following examples show what you must include in the XAResourceManager stanza in the qm.ini file on Linux and UNIX systems, and the equivalent information for Windows.

On Linux and UNIX (AIX, HP-UX, and Solaris):

1. Create the following symbolic links:

```
ln -s install_dir/sample/xatm/db2swit /var/mqm/exits/db2swit
ln -s install_dir/sample/xatm/db2swit64 /var/mqm/exits64/db2swit
```

2. Include the following code in the XAResourceManager stanza in the qm.ini file

```
XAResourceManager:
Name=DB2
SwitchFile=db2swit
XAOpenString=db=yourdatabase,uid=youruserid,pwd=yourpassword,toc=t
XACloseString=
ThreadOfControl=THREAD
```

On Windows, set the following values on the Resources page of the properties dialog for your WebSphere MQ queue manager. This example assumes that you have installed WebSphere Message Broker in the directory C:\WMQI:

```
SwitchFile: C:\wmqi\sample\xatm\db2swit
XAOpenString=db=yourdatabase,uid=youruserid,pwd=yourpassword,toc=t
ThreadOfControl=THREAD
```

Using DB2 with only 64-bit execution groups:

If you want to use DB2 in coordinated transactions:

- If you use DB2 Version 8, use the supplied switch file; a file called db2swit64 is provided for each supported platform. This file is installed as <Your install directory>/sample/xatm/db2swit64.
- If you prefer to build your own files, follow the instructions in the section "DB2 configuration" in the *WebSphere MQ System Administration Guide*. You can also find details of how to configure XA resource definitions in this book.

The following examples show what you must include in the XAResourceManager stanza in the qm.ini file on UNIX systems.

1. Create the following symbolic link:

```
ln -s install_dir/sample/xatm/db2swit64 /var/mqm/exits64/db2swit
```

2. Include the following code in the XAResourceManager stanza in the qm.ini file

```
XAResourceManager:
Name=DB2
SwitchFile=db2swit
XAOpenString=db=yourdatabase,uid=youruserid,pwd=yourpassword,toc=t
XACloseString=
ThreadOfControl=THREAD
```

Configuring databases for coordinated message flows with Oracle:

If your message flow interacts with an Oracle database, and you want to coordinate the updates made to the database with other actions within the message flow, configure your broker to manage these updates.

Before you start:

To complete this task, you must have completed the following task:

- “Configuring databases for coordinated message flows” on page 96

This topic describes how to use Oracle with a 32-bit broker, a 64-bit broker, and with either WebSphere MQ V5 or WebSphere MQ V6 and also describes the XAOpenString parameters that you need:

-
-
- “Using Oracle with WebSphere MQ V5 as the transaction coordinator”
- “Using Oracle with WebSphere MQ V6 as the transaction coordinator” on page 101
- “XAOpenString parameters” on page 102

Using Oracle with WebSphere MQ V5 as the transaction coordinator:

If you want to use Oracle in coordinated transactions:

1. Ensure that the user ID that is used to access the database and specified in the XAOpenString has the necessary Oracle privileges to access the DBA_PENDING_TRANSACTIONS view. You can grant the required access using the following Oracle SQLPLUS command:

```
grant select on DBA_PENDING_TRANSACTIONS to <userid>;
```

2. Use the switchfile supplied by WebSphere Message Broker. When you add the XAResourceManager configuration information for Oracle, specify:

- UKor8dtc20.so as the switchfile on AIX, Solaris and Linux (x86 platform).
- UKor8dtc20.sl as the switchfile on HP-UX
- UKor8dtc20.dll as the switchfile on Windows

3. Create the following symbolic links:

On AIX:

```
ln -s install_dir/merant/lib/libUKicu20.a /var/mqm/exits/libUKicu20.a
ln -s $ORACLE_HOME/lib/libclntsh.a /var/mqm/exits/libclntsh.a
```

On Solaris and Linux (x86 platform):

```
ln -s install_dir/merant/lib/libUKicu20.so /var/mqm/exits/libUKicu20.so
ln -s $ORACLE_HOME/lib/libclntsh.so /var/mqm/exits/libclntsh.so
```

On HP-UX:

```
ln -s install_dir/merant/lib/libUKicu20.sl /var/mqm/exits/libUKicu20.sl
ln -s $ORACLE_HOME/lib/libclntsh.sl /var/mqm/exits/libclntsh.sl
```

Refer to the information provided for the version of WebSphere MQ that you have installed:

- The following examples show what you must include in the XAResourceManager stanza in the qm.ini file on Linux and UNIX systems, and the equivalent information for Windows:

– On AIX:

```
XAResourceManager:
Name=OracleXA
SwitchFile=install_dir/merant/lib/UKor8dtc20.so
XAOpenString=ORACLE_XA+SQLNET=myserver+HostName=myhostname+PortNumber=myportnumber+Sid=mySID
+ACC=P/uid/passwd+sestm=100+threads=TRUE+DataSource=mydatasourcename+DB=mydatasourcename+K=2+
XACloseString=
ThreadOfControl=THREAD
```

– On HP-UX:

```
XAResourceManager:
Name=OracleXA
SwitchFile=install_dir/merant/lib/UKor8dtc20.sl
```

```
| XAOpenString=ORACLE_XA+SQLNET=myserver+HostName=myhostname+PortNumber=myportnumber+Sid=mySID
| +ACC=P/uid/passwd+sestm=100+threads=TRUE+DataSource=mydatasourcename+DB=mydatasourcename+K=2+
| XACloseString=
| ThreadOfControl=THREAD
```

– On Linux (x86 platform):

```
| XAResourceManager:
| Name=OracleXA
| SwitchFile=install_dir/merant/lib/UKor8dtc20.so
| XAOpenString=ORACLE_XA+SQLNET=myserver+HostName=myhostname+PortNumber=myportnumber+Sid=mySID
| +ACC=P/uid/passwd+sestm=100+threads=TRUE+DataSource=mydatasourcename+DB=mydatasourcename+K=2+
| XACloseString=
| ThreadOfControl=THREAD
```

– On Solaris:

```
| XAResourceManager:
| Name=OracleXA
| SwitchFile=install_dir/merant/lib/UKor8dtc20.so
| XAOpenString=ORACLE_XA+SQLNET=myserver+HostName=myhostname+PortNumber=myportnumber+Sid=mySID
| +ACC=P/uid/passwd+sestm=100+threads=TRUE+DataSource=mydatasourcename+DB=mydatasourcename+K=2+
| XACloseString=
| ThreadOfControl=THREAD
```

– On Windows, set the following values on the Resources page of the properties dialog for your WebSphere MQ queue manager (accessible from WebSphere MQ Services). This example assumes that you have installed WebSphere Message Broker in the directory C:\WMQI:

```
| SwitchFile: C:\WMQI\BIN\UKor8dtc20.dll
| XAOpenString=ORACLE_XA+SQLNET=myserver+HostName=myhostname+PortNumber=myportnumber+Sid=mySID
| +ACC=P/uid/passwd+sestm=100+threads=TRUE+DataSource=mydatasourcename+DB=mydatasourcename+K=2+
| ThreadOfControl: THREAD
```

Using Oracle with WebSphere MQ V6 as the transaction coordinator:

If you want to use Oracle in coordinated transactions:

1. Ensure that the user ID that is used to access the database and specified in the XAOpenString has the necessary Oracle privileges to access the DBA_PENDING_TRANSACTIONS view. You can grant the required access using the following Oracle SQLPLUS command:


```
grant select on DBA_PENDING_TRANSACTIONS to <userid>;
```
2. Use the switchfile supplied by WebSphere Message Broker. When you add the XAResourceManager configuration information for Oracle, specify:
 - UKor8dtc20.so as the switchfile on AIX and Solaris.
 - UKor8dtc20.sl as the switchfile on HP-UX.
3. Specify the hostname of the machine for the Oracle server, the port number on which it is listening, the Oracle Service ID (SID), the username and password that is to be used to access the database, and the name of the database to be coordinated.
4. Create the following symbolic link.

On AIX:

```
| ln -s install_dir/merant/lib/libUKicu20.a /var/mqm/exits/libUKicu20.a
| ln -s $ORACLE_HOME/lib/libcIntsh.a /var/mqm/exits/libcIntsh.a
| ln -s install_dir/DD64/lib/libUKicu20.a /var/mqm/exits64/libUKicu20.a
| ln -s install_dir/DD64/lib/UKoradtc20.so /var/mqm/exits64/UKor8dtc20.so
```

On HP-UX:

```
| ln -s install_dir/merant/lib/libUKicu20.sl /var/mqm/exits/libUKicu20.sl
| ln -s $ORACLE_HOME/lib/libcIntsh.sl /var/mqm/exits/libcIntsh.sl
| ln -s <Your install directory>/DD64/lib/libUKicu20.sl /var/mqm/exits64/libUKicu20.sl
| ln -s install_dir/DD64/lib/UKoradtc20.sl /var/mqm/exits64/UKor8dtc20.sl
```

On Linux (x86 platform):

```
| ln -s install_dir/merant/lib/libUKicu20.so /var/mqm/exits/libUKicu20.so
| ln -s $ORACLE_HOME/lib/libclntsh.so /var/mqm/exits/libclntsh.so
```

On Solaris:

```
| ln -s install_dir/merant/lib/libUKicu20.so /var/mqm/exits/libUKicu20.so
| ln -s $ORACLE_HOME/lib/libclntsh.so /var/mqm/exits/libclntsh.so
| ln -s install_dir/DD64/lib/libUKicu20.so /var/mqm/exits64/libUKicu20.so
| ln -s install_dir/DD64/lib/UKoradtc20.so /var/mqm/exits64/UKor8dtc20.so
```

The following examples show what you must include in the XAResourceManager stanza in the qm.ini file on UNIX systems, and the equivalent information for Windows:

- On AIX:

```
| XAResourceManager:
| Name=OracleXA
| SwitchFile=UKor8dtc20.so
| XAOpenString=ORACLE_XA+SQLNET=myserver+HostName=myhostname+PortNumber=myportnumber+Sid=mySID
| +ACC=P/uid/passwd+sestm=100+threads=TRUE+DataSource=mydatasourcename+DB=mydatasourcename+K=2+
| XACloseString=
| ThreadOfControl=THREAD
```

- On HP-UX:

```
| XAResourceManager:
| Name=OracleXA
| SwitchFile=UKor8dtc20.sl
| XAOpenString=ORACLE_XA+SQLNET=myserver+HostName=myhostname+PortNumber=myportnumber+Sid=mySID
| +ACC=P/uid/passwd+sestm=100+threads=TRUE+DataSource=mydatasourcename+DB=mydatasourcename+K=2+
| XACloseString=
| ThreadOfControl=THREAD
```

- On Linux (x86 platform):

```
| XAResourceManager:
| Name=OracleXA
| SwitchFile=UKor8dtc20.so
| XAOpenString=ORACLE_XA+SQLNET=myserver+HostName=myhostname+PortNumber=myportnumber+Sid=mySID
| +ACC=P/uid/passwd+sestm=100+threads=TRUE+DataSource=mydatasourcename+DB=mydatasourcename+K=2+
| XACloseString=
| ThreadOfControl=THREAD
```

- On Solaris:

```
| XAResourceManager:
| Name=OracleXA
| SwitchFile=UKor8dtc20.so
| XAOpenString=ORACLE_XA+SQLNET=myserver+HostName=myhostname+PortNumber=myportnumber+Sid=mySID
| +ACC=P/uid/passwd+sestm=100+threads=TRUE+DataSource=mydatasourcename+DB=mydatasourcename+K=2+
| XACloseString=
| ThreadOfControl=THREAD
```

- On Windows, set the following values on the Resources page of the properties dialog for your WebSphere MQ queue manager (accessible from WebSphere MQ Services). This example assumes that you have installed WebSphere Message Broker in the directory C:\WMQI:

```
| SwitchFile: UKor8dtc20.dll
| XAOpenString=ORACLE_XA+SQLNET=myserver+HostName=myhostname+PortNumber=myportnumber+Sid=mySID
| +ACC=P/uid/passwd+sestm=100+threads=TRUE+DataSource=mydatasourcename+DB=mydatasourcename+K=2+
| ThreadOfControl: THREAD
```

XAOpenString parameters:

Here is a list of the XAOpenString parameters that you need to include in the XAResourceManager stanza in the qm.ini file:

DataSource

The ODBC data source name for the database.

| **DB** The ODBC data source name for the database.

| **HostName**

| The name of the TCPIP host on which the Oracle database resides.

| **PortNumber**

| The TCPIP port on which the Oracle database is listening.

| **Sid** The Oracle System Identifier (SID) of the database.

| **SQLNET**

| The Oracle "Service name" that resolves to a "Connect Descriptor", for

| example through a mapping in the TSNAMES.ORA file.

Configuring databases for coordinated message flows with Sybase:

If your message flow interacts with a Sybase database, and you want to coordinate the updates made to the database with other actions within the message flow, configure your broker to manage these updates.

Before you start:

To complete this task, you must have completed the following task:

- "Configuring databases for coordinated message flows" on page 96

Using Sybase with only 32-bit execution groups:

If you want to use Sybase in coordinated transactions, follow the general instructions in the section called "Sybase configuration" in the *WebSphere MQ System Administration* book, and use these with the instructions given here. WebSphere Message Broker supports only Sybase Version 12_5.

1. Ensure that the DataSourceUserId specified when the broker is created is a user ID that has been granted the Sybase role of dtm_tm_role.
2. Use the switchfile supplied by WebSphere Message Broker. When you add the XAResourceManager configuration information for Sybase, specify:
 - UKasedtc20.so as the switchfile on AIX, Solaris, and Linux (x86 platform)
 - UKasedtc20.s1 as the switchfile on HP-UX
 - UKase20.dll as the switchfile on Windows
3. Specify the ODBC DSN name of your database in the -N parameter of the XAOpenString, the network address of your database in the -A parameter, the user name used for access in the -U parameter, and the password for that user in the -P parameter.
4. Create the following symbolic links:

On AIX:

```
ln -s install_dir/merant/lib/libUKicu20.a /var/mqm/exits/libUKicu20.a
ln -s install_dir/merant/lib/UKase20.so /var/mqm/exits/UKase20.so
ln -s install_dir/merant/lib/UKasedtc20.so /var/mqm/exits/UKasedtc20.so
```

On Solaris and Linux (x86 platform):

```
ln -s install_dir/merant/lib/libUKicu20.so /var/mqm/exits/libUKicu20.so
ln -s install_dir/merant/lib/UKase20.so /var/mqm/exits/UKase20.so
ln -s install_dir/merant/lib/UKasedtc20.so /var/mqm/exits/UKasedtc20.so
```

On HP-UX:

```
ln -s install_dir/merant/lib/libUKicu20.s1 /var/mqm/exits/libUKicu20.s1
ln -s install_dir/merant/lib/UKase20.s1 /var/mqm/exits/UKase20.s1
ln -s install_dir/merant/lib/UKasedtc20.s1 /var/mqm/exits/UKasedtc20.s1
```

Refer to the information provided for the version of WebSphere MQ that you have installed:

- For coordination by WebSphere MQ V5:

The following examples show what you must include in the XAResourceManager stanza in the qm.ini file on UNIX systems, and the equivalent Windows information. In each example:

- N is the ODBC data source name.
- A is the TCPIP host and port on which the Sybase ASE server resides.
- U represents the user login.
- P represents the user ID's password.
- On AIX:

```
XAResourceManager:  
  Name=SYBASEXA  
  SwitchFile=<Your install directory>/merant/lib/UKasedtc20.so  
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2  
  XACloseString=  
  ThreadOfControl=THREAD
```

- On HP-UX:

```
XAResourceManager:  
  Name=SYBASEXA  
  SwitchFile=<Your install directory>/merant/lib/UKasedtc20.sl  
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2  
  XACloseString=  
  ThreadOfControl=THREAD
```

- On Linux (x86 platform):

```
XAResourceManager:  
  Name=SYBASEXA  
  SwitchFile=<Your install directory>/merant/lib/UKasedtc20.so  
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2  
  XACloseString=  
  ThreadOfControl=THREAD
```

- On Solaris:

```
XAResourceManager:  
  Name=SYBASEXA  
  SwitchFile=<Your install directory>/merant/lib/UKasedtc20.so  
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2  
  XACloseString=  
  ThreadOfControl=THREAD
```

- On Windows, set the following values on the Resources page of the properties dialog for your WebSphere MQ queue manager (accessible from WebSphere MQ Services). This example assumes you have installed WebSphere Message Broker in the directory C:\WMQI:

```
SwitchFile: C:\WMQI\BIN\UKase20.dll  
XAOpenString: -NSYBASEDB -A<YourServerName,YourPortNumber> -WWinsock -Uuid -Ppwd -K2  
XACloseString:  
ThreadOfControl: THREAD
```

- For coordination by WebSphere MQ V6:

The following examples show what you must include in the XAResourceManager stanza in the qm.ini file on UNIX systems, and the equivalent Windows information. In each example:

- N is the ODBC data source name.
- A is the TCPIP host and port on which the Sybase ASE server resides.
- U represents the user login.
- P represents the user ID's password.

- On AIX:


```
XAResourceManager:
  Name=SYBASEXA
  SwitchFile=UKasedtc20.so
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2
  XACloseString=
  ThreadOfControl=THREAD
```
- On HP-UX:


```
XAResourceManager:
  Name=SYBASEXA
  SwitchFile=UKasedtc20.s1
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2
  XACloseString=
  ThreadOfControl=THREAD
```
- On Linux (x86 platform):


```
XAResourceManager:
  Name=SYBASEXA
  SwitchFile=UKasedtc20.so
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2
  XACloseString=
  ThreadOfControl=THREAD
```
- On Solaris:


```
XAResourceManager:
  Name=SYBASEXA
  SwitchFile=UKasedtc20.so
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2
  XACloseString=
  ThreadOfControl=THREAD
```
- On Windows, set the following values on the Resources page of the properties dialog for your WebSphere MQ queue manager (accessible from WebSphere MQ Services). This example assumes you have installed WebSphere Message Broker in the directory C:\WMQI:


```
SwitchFile: C:\WMQI\BIN\UKase20.d11
XAOpenString: -NSYBASEDB -A<YourServerName,YourPortNumber> -WWinsock -Uuid -Ppwd -K2
XACloseString:
ThreadOfControl: THREAD
```

Using Sybase with only 64-bit execution groups:

If you want to use Sybase in coordinated transactions, follow the general instructions in the section called "Sybase configuration" in the *WebSphere MQ System Administration* book, and use these with the instructions given here. WebSphere Message Broker supports only Sybase Version 12_5.

1. Ensure that the DataSourceUserId specified when the broker is created is a user ID that has been granted the Sybase role of dtm_tm_role.
2. Use the switchfile supplied by WebSphere Message Broker. When you add the XAResourceManager configuration information for Sybase, specify:
 - UKasedtc20.so as the switchfile on AIX and Solaris.
 - UKasedtc20.s1 as the switchfile on HP-UX
3. Specify the ODBC DSN name of your database in the -N parameter of the XAOpenString, the network address of your database in the -A parameter, the user name used for access in the -U parameter, and the password for that user in the -P parameter.
4. Create the following symbolic links:

On AIX:

```
ln -s install_dir/DD64/lib/libUKicu20.a /var/mqm/exits64/libUKicu20.a
ln -s install_dir/DD64/lib/UKase20.so /var/mqm/exits64/UKase20.so
ln -s install_dir/DD64/lib/UKasedtc20.so /var/mqm/exits64/UKasedtc20.so
```

On HP-UX:

```
ln -s install_dir/DD64/lib/libUKicu20.sl /var/mqm/exits64/libUKicu20.sl
ln -s install_dir/DD64/lib/UKase20.sl /var/mqm/exits64/UKase20.sl
ln -s install_dir/DD64/lib/UKasedtc20.sl /var/mqm/exits64/UKasedtc20.sl
```

On Solaris:

```
ln -s install_dir/DD64/lib/libUKicu20.so /var/mqm/exits64/libUKicu20.so
ln -s install_dir/DD64/lib/UKase20.so /var/mqm/exits64/UKase20.so
ln -s install_dir/DD64/lib/UKasedtc20.so /var/mqm/exits64/UKasedtc20.so
```

The following examples show what you must include in the XAResourceManager stanza in the qm.ini file on UNIX systems, and the equivalent Windows information. In each example:

- N is the ODBC data source name.
- A is the TCPIP host and port on which the Sybase ASE server resides.
- U represents the user login.
- P represents the user ID's password.
- On AIX:

```
XAResourceManager:
  Name=SYBASEXA
  SwitchFile=UKasedtc20.so
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2
  XACloseString=
  ThreadOfControl=THREAD
```

- On HP-UX:

```
XAResourceManager:
  Name=SYBASEXA
  SwitchFile=UKasedtc20.sl
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2
  XACloseString=
  ThreadOfControl=THREAD
```

- On Solaris:

```
XAResourceManager:
  Name=SYBASEXA
  SwitchFile=UKasedtc20.so
  XAOpenString=-NSYBASEDB -A<YourServerName,YourPortNumber> -Uuid -Ppwd -K2
  XACloseString=
  ThreadOfControl=THREAD
```

Configuring WebSphere MQ and RRS for coordinated message flows

If you have configured a message flow and the databases it interacts with to coordinate the processing it performs, you might also need to configure WebSphere MQ and RRS to support this.

Before you start:

To complete this task, you must have completed the following task:

- “Configuring message flow transactions” on page 93

The details of the actions that you might need to complete are described in the WebSphere MQ library.

For an example of how to configure WebSphere MQ as the transaction manager, see the Error Handling sample.

Configuring message flows for data conversion

If you exchange messages between applications that run on systems that are incompatible in some way, you can configure your system to provide data conversion as the message passes through the broker. Data conversion might be necessary if either of the following two values are different on the sending and receiving systems:

1. **CCSID.** The Coded Character Set Identifier refers to a set of coded characters and their code point assignments. WebSphere Message Broker can process and construct application messages in any code page for which WebSphere MQ provides conversion to and from Unicode, on all operating systems. For more information about code page support, see the *WebSphere MQ Application Programming Reference*.

This behavior might be affected by the use of other products in conjunction with WebSphere Message Broker. Check the documentation for other products, including any databases that you use, for further code page support information.

2. **Encoding.** This defines the way in which a machine encodes numbers, that is binary integers, packed-decimal integers, and floating point numbers. Numbers that are represented as characters are handled in the same way as all other string data.

If the native CCSID and encoding on the sending and receiving systems are the same, you do not need to invoke data conversion processes.

WebSphere Message Broker and WebSphere MQ provide data conversion facilities to support message exchange between unlike systems. Your choice of which facilities to use depends on the characteristics of the messages that are processed by the message flow:

- Messages that contain text only
- Message that include numerics
- Messages that are self-defining

Messages that contain text only

Read this section if your messages are WebSphere MQ messages that contain all text (character data or string). If WebSphere MQ supports the systems on which both sending and receiving applications are running for data conversion, use WebSphere MQ facilities. This provides the most efficient data conversion option.

The default behavior of WebSphere MQ is to put messages to queues specifying the local system CCSID and encoding. Applications issuing MQGET can request that the queue manager provides conversion to their local CCSID and encoding as part of get processing.

To use this option:

1. Design messages to be text-only. If you are using COBOL, move numeric fields to USAGE DISPLAY to put them into string form.
2. Set the Format field in the MQMD to MQFMT_STRING (value MQSTR).

3. Issue MQGET with MQGMO_CONVERT in the receiving application. If you prefer, you can convert when the message is received by the broker, by setting the *Convert* property of the MQInput node to yes (by selecting the check box).

If you require more sophisticated data conversion than WebSphere MQ provides in this way (for example, to an unsupported code page), use WebSphere MQ data conversion exits. For more information about these, see the *WebSphere MQ Application Programming Reference*.

Messages that include numerics

Read this section if your messages include numeric data, or are text only but are not WebSphere MQ messages. If these messages can be predefined (that is, their content and structure is known and predictable), use the facilities provided by WebSphere Message Broker and the MRM.

All application messages are handled by the broker in Unicode, to which they are converted on input, and from which they are converted on output. You can configure message flows to influence the way in which output messages are constructed.

To use this option:

1. Define the output message in the MRM domain. You can create this definition in one of the following ways:
 - Import an external message definition (for example a C header or COBOL copybook).
 - Create the message model in the message definition editor.
2. Configure a message flow to receive and process this message:
 - a. If you include an MQInput node, do not request conversion by this node.
 - b. Include a Compute node in the message flow to create the output message with the required content:
 - If the output message is a WebSphere MQ message, code ESQL in the Compute node to set the CCSID and encoding for the target system in the MQMD.
For example, to set values for a target z/OS system running with CCSID of 37 and encoding of 785:

```
SET OutputRoot.MQMD.CodedCharSetId = 37;  
SET OutputRoot.MQMD.Encoding = 785;
```
 - If the output message is not a WebSphere MQ message, code ESQL in the Compute node to set the CCSID and encoding for the target system in the Properties folder.

Messages that are self-defining

Read this section if your messages are self-defining.

Self-defining messages are supported in the XML and JMS domains. These messages are all text and can be handled by WebSphere MQ, if they originate from, or are destined for, WebSphere MQ applications. If not, use WebSphere Message Broker facilities by setting the CCSID and Encoding fields in the Properties folder in the message when it passes through a Compute node.

Ensuring that messages are not lost

It is important to safeguard that messages that flow through your broker domain. This is true of both application-generated messages and those used internally for inter-component communication. Messages used internally between components always use the WebSphere MQ protocol. Application messages can use all supported transport protocols.

For application and internal messages travelling across WebSphere MQ, two techniques protect against message loss:

- Message persistence
If a message is persistent, WebSphere MQ ensures that it is not lost when a failure occurs, by copying it to disk.
- Syncpoint control
An application can request that a message is processed in a synchronized unit-of-work (UOW)

For more information about how to use these options, refer to the *WebSphere MQ System Administration Guide*.

Internal messages

WebSphere Message Broker components use WebSphere MQ messages to communicate events and data between broker processes and subsystems, and the Configuration Manager and User Name Server. The components ensure that the WebSphere MQ features are exploited to protect against message loss. You do not need to take any additional steps to configure WebSphere MQ or WebSphere Message Broker to protect against loss of internal messages.

Application messages

If delivery of application messages is critical, you must design application programs and the message flows that they use to ensure that messages are not lost. The techniques used depend on the protocol used by the applications.

WebSphere MQ Enterprise Transport and WebSphere MQ Mobile Transport

If you are using the built-in input nodes that accept messages across the WebSphere MQ or WebSphere MQ Everyplace protocols, you can use the following guidelines and recommendations:

- Using persistent messages

WebSphere MQ messaging products provide message persistence, which defines the longevity of the message in the system and guarantees message integrity. Nonpersistent messages are lost in the event of system or queue manager failure. Persistent messages are always recovered if a failure occurs.

You can control message persistence in the following ways:

- Program your applications that put messages to a queue using the MQI or AMI to indicate that the messages are persistent.
- Define the input queue with message persistence as the default setting.
- Configure the output node to handle persistent messages.
- Program your subscriber applications to request message persistence.

When an input node reads a message is read from an input queue, the default action is to use the persistence defined in the WebSphere MQ

message header (MQMD), that has been set either by the application creating the message, or by the default persistence of the input queue. The message retains this persistence throughout the message flow, unless it is changed in a subsequent message processing node.

You can override the persistence value of each message when the message flow terminates at an output node. This node has a property that allows you to specify the message persistence of each message when it is put to the output queue, either as the required value, or as a default value. If you specify the default, the message takes the persistence value defined for the queues to which the messages are written.

If a message passes through a Publication node, the persistence of messages sent to subscribers is determined by the subscribers' registration options. If a subscriber has requested persistent message delivery, and is authorized to do so by explicit or implicit (inherited) ACL, the message is delivered persistently regardless of its existing persistence property. Also, if the user has requested nonpersistent message delivery, the message is delivered nonpersistent regardless of its existing persistence property.

If a message flow creates a new message (for example, in a Compute node), the persistence in the MQMD of the new message is copied from the persistence in the MQMD of the incoming message.

- **Processing messages under syncpoint control**

The default action of a message flow is to process incoming messages under syncpoint in a broker-controlled transaction. This means that a message that fails to be processed for any reason is backed out by the broker. Because it was received under syncpoint, the failing message is reinstated on the input queue and can be processed again. If the processing fails, the error handling procedures that are in place for this message flow (defined either by how you have configured the message flow, or by the broker) are executed.

For full details of input node processing, see "Managing errors in the input node" on page 114.

WebSphere MQ Telemetry Transport

If you are using the built-in input node SCADAInput that accepts messages from telemetry devices across the MQIsdp protocol, this protocol does not have a concept of queues. Clients connect to a SCADAInput node by specifying the port number on which the node is listening. Messages are sent to clients using a `clientId`. However, you can specify a maximum QoS (Quality of Service) within a SCADA subscription message, which is similar to persistence:

- **QoS0** Nonpersistent.
- **QoS1** Persistent, but might be delivered more than once
- **QoS2** Once and once only delivery

If a persistent SCADA message is published, it might be downgraded to the highest level that the client can accept. In some circumstances this might mean that the message becomes nonpersistent.

WebSphere MQ Real-time Transport, WebSphere MQ Multicast Transport, and WebSphere MQ Web Services Transport

If you are using the built-in input nodes Real-timeInput and Real-timeOptimizedFlow that accept messages from JMS and multicast applications, or the HTTP Input and HTTPRequest nodes that accept messages from Web services applications, no facilities are available to

protect against message loss. You can, however, provide recovery procedures by configuring the message flow to handle its own errors.

Other transports and protocols

If you have created your own user-defined input nodes that receive messages from another transport protocol, you must rely on the support provided by that transport protocol, or provide your own recovery procedures.

Handling errors in message flows

The broker provides basic error handling for all your message flows. If basic processing is not sufficient, and you want to take specific action in response to certain error conditions and situations, you can enhance your message flows to provide your own error handling. For example, you might design a message flow that expects certain errors that you want to process in a particular way, or a flow that updates a database and must roll back those updates if other processing does not complete successfully.

The options that you can use to do this are quite complex in some cases. The options that are provided for MQInput and TimeoutNotification nodes are extensive because these nodes deal with persistent messages and transactions. MQInput is also affected by configuration options for WebSphere MQ.

Because you can decide to handle different errors in different ways, there are no fixed procedures to describe. This section provides information about the principles of error handling, and the options that are available, and you must decide what combination of choices that you need in each situation based on the details that are provided in this section.

You can choose one or more of these options in your message flows:

- Connect the failure terminal of any node to a sequence of nodes that processes the node's internal exception (the fail flow).
- Connect the catch terminal of the input node or a TryCatch node to a sequence of nodes that processes exceptions that are generated beyond it (the catch flow).
- Insert one or more TryCatch nodes at specific points in the message flow to catch and process exceptions that are generated by the flow connected to the try terminal.
- Include a Throw node, or code an `ESQL THROW` statement, to generate an exception.
- If you are using aggregation, connect the catch terminal of the AggregateReply node to process aggregation exceptions.
- Ensure that all messages received by an MQInput node are processed within a transaction, or are not.
- Ensure that all messages received by an MQInput node are persistent, or are not.

If you include user-defined nodes in your message flow, you must refer to the information provided with the node to understand how you might handle errors with these nodes. The descriptions in this section cover only the built-in nodes.

When you design your error handling approach, consider the following factors:

- Most of the built-in nodes have failure terminals. The exceptions are AggregateControl, AggregateRequest, Input, Label, Output, Passthrough, Publication, Real-timeInput, Real-timeOptimizedFlow, Throw, Trace, and TryCatch.

When an exception is detected within a node, the message and the exception information are propagated to the node's failure terminal. If the node does not have a failure terminal, or it is not connected, the broker throws an exception and returns control to the closest previous node that can process it. This might be a TryCatch node, an AggregateReply node, or the input node.

If an MQInput node detects an internal error, its behavior is slightly different; if the failure terminal is not connected, it attempts to put the message to the input queue's backout requeue queue, or (if that is not defined) to the dead letter queue of the broker's queue manager.

For more information, see the following:

- "MQInput error handling" on page 115

- A small number of built-in nodes have catch terminals. These are AggregateReply, HTTPInput, MQInput, SCADAInput, JMSInput, JMSOutput, TimeoutNotification, and TryCatch.

A message is propagated to a catch terminal only if it has first been propagated beyond the node (for example, to the nodes connected to the out terminal).

- When a message is propagated to the failure or catch terminal, the node creates and populates a new ExceptionList with an exception that represents the error that has occurred. The ExceptionList is propagated as part of the message tree.
- The MQInput and TimeoutNotification nodes have additional processing for transactional messages (other input nodes do not handle transactional messages).

For more information, see the following topics:

- "MQInput error handling" on page 115
- "TimeoutNotification error handling" on page 119

- If you include a Trace node that specifies \$Root or \$Body, the complete message is parsed. This might generate parser errors that are not otherwise detected.

The general principles of error handling are:

- If you connect the catch terminal of the input node, you are indicating that the flow handles all exceptions that are generated anywhere in the out flow. The broker performs no rollback and takes no action unless there is an exception on the catch flow. If you want any rollback action after an exception has been raised and caught, you must provide this in the catch flow.
- If you do not connect the catch terminal of the MQInput or HTTPInput node, you can connect the failure terminal and provide a fail flow to handle exceptions generated by the node. The fail flow is invoked immediately when an exception occurs in the node.

The fail flow is also invoked if an exception is generated beyond the MQInput node (in either out or catch flows), the message is transactional, and the reinstatement of the message on the input queue causes the backout count to reach the backout threshold.

The HTTPInput and SCADAInput nodes do not propagate the message to the failure terminal if an exception is generated beyond the node and you have not connected the catch terminal.

- If a node propagates a message to a catch flow, and another exception occurs that returns control to the same node again, the node handles the message as though the catch terminal is not connected.

- If you do not connect either failure or catch terminals of the input node, the broker provides default processing (which varies with the type of input node).
- If you need a more comprehensive error and recovery approach, include one or more TryCatch nodes to provide more localized areas of error handling.
- If you have a common procedure for handling particular errors, you might find it appropriate to create a subflow that includes the sequence of nodes required. Include this subflow wherever you need that action to be taken.

For more information, see the following topics:

- “Connecting failure terminals”
- “Managing errors in the input node” on page 114
- “Catching exceptions in a TryCatch node” on page 120

If your message flows include database updates, the way in which you configure the nodes that interact with those databases can also affect the way that errors are handled:

- You can specify whether updates are committed or rolled back. You can set the node property *Transaction* to specify whether database updates are committed or rolled back with the message flow (option *Automatic*) or committed or rolled back when the node itself terminates (option *Commit*). You must ensure that the combination of these property settings and the message flow error processing give the correct result.
- You can specify how database errors are handled. You can set the properties *Treat warnings as errors* and *Throw exception on database error* to change the default behavior of database error handling.

For more information about coordinated database updates, see “Configuring message flow transactions” on page 93.

Message flows for aggregation involve additional considerations that are not discussed in this section; these are described in “Handling exceptions in aggregation flows” on page 400.

The Error Handler sample demonstrates how to use an error handling routine to trap information about errors and to store that information in a database. The error handling routine is a subflow that you can add, unchanged, to any message flow. The sample also demonstrates how to configure message flows to control transactionality; in particular, the use of globally coordinated transactions to ensure overall data integrity.

Connecting failure terminals

When a node that has a failure terminal detects an internal error, it propagates the message to that terminal. If it does not have a failure terminal, or if you have not connected the failure terminal, the broker generates an exception.

The nodes sometimes generate errors that you can predict, and it is in these cases that you might want to consider connecting the failure terminal to a sequence of nodes that can take sensible actions in response to the expected errors.

Examples of expected errors are:

- Temporary errors when the input node retrieves the message.
- Validation errors detected by an MQInput, Compute, or Mapping node.

- Messages with an internal or format error that cannot be recognized or processed by the input node.
- Acceptable errors when a node accesses a database, and you choose not to configure the node to handle those errors.
- ESQL errors during message flow development (some ESQL errors cannot be detected by the editor, but are recognized only by the broker; these cause an exception if you have not connected the failure terminal. You can remove the fail flow when you have completely tested the runtime ESQL code).

You can also connect the failure terminal if you do not want WebSphere MQ to retry a message or put it to a backout or dead letter queue.

Managing errors in the input node

When you design your message flow, consider which terminals on the input node to connect:

- If the node detects an error, it always propagates the message to the failure terminal if the node has one and if you have connected a fail flow.
- If you connect the catch terminal (if the node has one), this indicates that you want to handle all exceptions that are generated in the out flow. This handles errors that can be expected in the out flow. The broker does not take any action unless there is an exception on the catch flow and the message is transactional. Connect the failure terminal to handle this case if you choose.
- If you do not connect the catch terminal, or the node does not have a catch terminal, the broker provides default processing. This depends on the node and whether the message is transactional. Processing for non-transactional messages is described in this topic. Refer to “MQInput error handling” on page 115, and “TimeoutNotification error handling” on page 119 for details of how these nodes handle transactional messages (other input nodes do not support transactional messages).

All input nodes process non-transactional, non-persistent messages. The built-in input nodes handle failures and exceptions associated with these messages in this way:

- If the node detects an internal error:
 - If you have not connected the failure terminal, the node logs the error in the local error log and discards the message.
The Real-timeInput and Real-timeOptimizedFlow nodes retry once before they discard the message; that is, they retrieve the message again and attempt to process it.
 - If you have connected the failure terminal, you are responsible for handling the error in the fail flow. The broker creates a new ExceptionList to represent the error and this is propagated to the failure terminal as part of the message tree, but neither the node nor the broker provide any further failure processing.
- If the node has successfully propagated the message to the out terminal and a later exception results in the message being returned to the input node:
 - If you have not connected the catch terminal or the node does not have a catch terminal, the node logs the error in the local error log and discards the message.
 - If you have connected the catch terminal, you are responsible for handling the error in the catch flow. The broker creates a new ExceptionList to represent

the error and this is propagated to the catch terminal as part of the message tree, but neither the node nor the broker provide any further exception processing.

- If the node has already propagated the message to the catch terminal and an exception is thrown in the catch flow:
 - If you have not connected the failure terminal, or the input node does not have a failure terminal, the node logs the error in the local error log and discards the message.
 - If you have connected the failure terminal, you are responsible for handling the error in the fail flow. The broker creates a new ExceptionList to represent the error and this is propagated to the failure terminal as part of the message tree, but neither the node nor the broker provide any further failure processing.

The HTTPInput and SCADAInput nodes do not propagate the message to the failure terminal if an exception is generated in the catch flow. The node logs the error in the local error log and discards the message.

- If the node has propagated the message to the failure terminal and an exception is thrown in the fail flow, the node logs the error in the local error log and discards the message.

In every situation in which it discards the message, the HTTPInput node waits until the time specified by the node property *Maximum client wait time* expires, and returns an error to the Web services client.

This action is summarized in the table below:

Error event	Failure terminal connected	Failure terminal not connected	Catch terminal connected	Catch terminal not connected
Node detects internal error	Fail flow handles the error	Node logs the error and discards the message	Not applicable	Not applicable
Node propagates message to out terminal, exception occurs in out flow	Not applicable	Not applicable	Catch flow handles the error	Node logs the error and discards the message
Node propagates message to catch terminal, exception occurs in catch flow	Fail flow handles the error (not HTTPInput or SCADAInput)	Node logs the error and discards the message	Not applicable	Not applicable
Node propagates message to failure terminal, exception occurs in fail flow	Not applicable	Not applicable	Node logs the error and discards the message	Node logs the error and discards the message

MQInput error handling:

The MQInput node takes the following actions when it handles errors with persistent and transactional messages. Errors encountered with non-transactional messages are handled as described in “Managing errors in the input node” on page 114.

- The MQInput node detects an internal error in the following situations:
 - A message validation error occurs when the associated message parser is initialized.
 - A warning is received on an MQGET.
 - The backout threshold is reached when the message is rolled back to the input queue.
- If the MQInput node detects an internal error, one of the following actions occur:
 - If you have not connected the Failure terminal, the MQInput node attempts to put the message to the input queue's backout requeue queue, or (if that is not defined) to the dead letter queue of the broker's queue manager. If the put attempt fails, the message is rolled back to the input queue. The MQInput node writes the original error and the MQPUT error to the local error log. The MQInput node now invokes the retry logic, described in "Retry processing" on page 117.
 - If you have connected the Failure terminal, you are responsible for handling the error in the flow connected to the Failure terminal. The broker creates a new ExceptionList to represent the error and this is propagated to the Failure terminal as part of the message tree, but neither the MQInput node nor the broker provide any further failure processing.
- If the MQInput node has successfully propagated the message to the out terminal and an exception is thrown in the out flow, the message is returned to the MQInput node:
 - If you have not connected the Catch terminal, the message is rolled back to the input queue. The MQInput node writes the error to the local error log and invokes the retry logic, described in "Retry processing" on page 117.
 - If you have connected the Catch terminal, you are responsible for handling the error in the flow connected to the Catch terminal. The broker creates a new ExceptionList to represent the error and this is propagated to the Catch terminal as part of the message tree, but neither the MQInput node nor the broker provide any further failure processing.
- If the MQInput node has already propagated the message to the Catch terminal and an exception is thrown in the flow connected to the Catch terminal, the message is returned to the MQInput node:
 - The MQInput node writes the error to the local error log.
 - The message is rolled back to the input queue.
- If the MQInput node has already propagated the message to the Failure terminal and an exception is thrown in the flow connected to the Failure terminal, the message is returned to the MQInput node and rolled back to the input queue. The MQInput node writes the error to the local error log and invokes the retry logic, described in "Retry processing" on page 117. The message is not propagated to the Catch terminal, even if that is connected.

This action is summarized in the table below:

Error event	Failure terminal connected	Failure terminal not connected	Catch terminal connected	Catch terminal not connected
Node detects internal error	Flow connected to the Failure terminal handles the error	Message put to alternative queue; node retries if the put fails	Not applicable	Not applicable

Error event	Failure terminal connected	Failure terminal not connected	Catch terminal connected	Catch terminal not connected
Node propagates message to out terminal, exception occurs in out flow	Not applicable	Not applicable	Flow connected to the Catch terminal handles the error	Node retries
Node propagates message to Catch terminal, exception occurs in flow connected to the Catch terminal	Error logged, message rolled back	Error logged, message rolled back	Not applicable	Not applicable
Node propagates message to Failure terminal, exception occurs in flow connected to the Failure terminal	Not applicable	Not applicable	Node retries	Node retries

Retry processing:

The node attempts retry processing when a message is rolled back to the input queue. It checks whether the message has been backed out before, and if it has, whether the backout count has reached (equalled) the backout threshold. The backout count for each message is maintained by WebSphere MQ in the MQMD.

You specify (or allow to default to 0) the backout threshold attribute *BOTHRESH* when you create the queue. If you accept the default value of 0, the node increases this to 1. The node also sets the value to 1 if it cannot detect the current value. This means that if a message has not been backed out before, it is backed out and retried at least once.

1. If the node has propagated a message to the out terminal many times following repeated failed attempts in the out flow, and the number of retries has reached the backout threshold limit, it attempts to propagate the message through the Failure terminal if that is connected. If you have not connected the Failure terminal, the node attempts to put the message to another queue.
If a failure occurs beyond the Failure terminal, further retries are made until the backout count field in the MQMD reaches twice the backout threshold set for the input queue. When this limit is reached, the node attempts to put the message to another queue.
2. If the backout threshold has not been reached, the node gets the message from the queue again. If this fails, this is handled as an internal error (described above). If it succeeds, the node propagates the message to the out flow.
3. If the backout threshold has been reached:
 - If you have connected the Failure terminal, node propagates the message to that terminal. You must handle the error on the flow connected to the Failure terminal.
 - If you have not connected the Failure terminal, the node attempts to put the message on an available queue, in order of preference:
 - a. The message is put on the input queue's backout requeue name (queue attribute *BOQNAME*), if one is defined.

- b. If the backout queue is not defined, or it cannot be identified by the node, the message is put on the dead letter queue (DLQ), if one is defined. (If the broker's queue manager has been defined by the **mqsicreatebroker** command, a DLQ with a default name of `SYSTEM.DEAD.LETTER.QUEUE` has been defined and is enabled for this queue manager.)
 - c. If the message cannot be put on either of these queues because there is an MQPUT error (including queue does not exist), or because they cannot be identified by the node, it cannot be handled safely without risk of loss. The message cannot be discarded, therefore the message flow continues to attempt to backout the message. It records the error situation by writing errors to the local error log. A second indication of this error is the continual incrementing of the *BackoutCount* of the message in the input queue.
If this situation has occurred because neither queue exists, you can define one of the backout queues mentioned above. If the condition preventing the message from being processed has cleared, you can temporarily increase the value of the *BOTHRESH* attribute. This forces the message through normal processing.
4. If twice the backout threshold has been reached or exceeded, the node attempts to put the message on an available queue, in order of preference, as defined in the previous step.

Handling message group errors:

WebSphere MQ supports message groups. You can specify that a message belongs to a group and its processing is then completed with reference to the other messages in the group (that is, either all messages are committed or all messages are rolled back). When you send grouped messages to a broker, this condition is upheld if you have configured the message flow correctly, and errors do not occur during group message processing.

To configure the message flow to handle grouped messages correctly, follow the actions described in the "MQInput node" on page 593. However, correct processing of the message group cannot be guaranteed if an error occurs while one of the messages is being processed.

If you have configured the MQInput node as described, under normal circumstances all messages in the group are processed in a single unit of work which is committed when the last message in the group has been successfully processed. However, if an error occurs before the last message in the group is processed, the unit of work that includes the messages up to and including the message that generates the error is subject to the error handling defined by the rules documented here, which might result in the unit of work being backed out.

However, any of the remaining messages within the group might be successfully read and processed by the message flow, and therefore are handled and committed in a new unit of work. A commit is issued when the last message is encountered and processed. Therefore if an error occurs within a group, but not on the first or last message, it is possible that part of the group is backed out and another part committed.

If your message processing requirements demand that this situation is handled in a particular way, you must provide additional error handling to handle errors within message groups. For example, you could record the failure of the message group

within a database, and include a check on the database when you retrieve each message, forcing a rollback if the current group has already encountered an error. This would ensure that the whole group of messages is backed out and not processed unless all are successful.

TimeoutNotification error handling: The TimeoutNotification node takes the following actions when it handles errors with transactional messages. Errors encountered with non-transactional messages are handled as described in “Managing errors in the input node” on page 114.

- If the TimeoutNotification node detects an internal error, one of the following actions occur:
 - If you have not connected the Failure terminal the following happens:
 1. The TimeoutNotification node writes the error to the local error log.
 2. The TimeoutNotification node repeatedly tries to process the request until the problem has been resolved.
 - If you have connected the Failure terminal, you are responsible for handling the error in the flow connected to the Failure terminal. The broker creates a new ExceptionList to represent the error and this is propagated to the Failure terminal as part of the message tree, but neither the TimeoutNotification node nor the broker provide any further failure processing. The message is written to the Failure terminal as part of the same transaction, and if the failure flow handles the error successfully the transaction is committed.
- If the TimeoutNotification node has successfully propagated the message to the Out terminal and an exception is thrown in the flow connected to the Out terminal, the message is returned to the TimeoutNotification node. The TimeoutNotification node writes the error to the local error log and does one of the following:
 - If you have not connected the Catch terminal, the TimeoutNotification node tries to process the message again until the problem is resolved.
 - If you have connected the Catch terminal, you are responsible for handling the error in the flow connected to the Catch terminal. The broker creates a new ExceptionList to represent the error and this is propagated to the Catch terminal as part of the message tree, but neither the TimeoutNotification node nor the broker provide any further failure processing. The message is written to the Catch terminal as part of the same transaction, and if the flow connected to the Catch terminal handles the error successfully the transaction is committed.
- If the TimeoutNotification node has already propagated the message to the Catch terminal and an exception is thrown in the flow connected to the Catch terminal, the message is returned to the TimeoutNotification node. The TimeoutNotification node writes the error to the local error log and tries to process the message again.
- If the TimeoutNotification node has already propagated the message to the Failure terminal and an exception is thrown in the flow connected to the Failure terminal, the message is returned to the TimeoutNotification node. The TimeoutNotification node writes the error to the local error log and tries to process the message again. The message is not propagated to the Catch terminal, even if that is connected.

This action is summarized in the table below:

Error event	Failure terminal connected	Failure terminal not connected	Catch terminal connected	Catch terminal not connected
Node detects internal error	Flow connected to the Failure terminal handles the error	Error logged, node retries	Not applicable	Not applicable
Node propagates message to out terminal, exception occurs in out flow	Not applicable	Not applicable	Flow connected to the Catch terminal handles the error	Error logged, node retries
Node propagates message to Catch terminal, exception occurs in flow connected to the Catch terminal	Error logged, node retries	Error logged, node retries	Not applicable	Not applicable
Node propagates message to Failure terminal, exception occurs in flow connected to the Failure terminal	Not applicable	Not applicable	Error logged, node retries	Error logged, node retries

Catching exceptions in a TryCatch node

You can design a message flow to catch exceptions before they are returned to the input node. Within a single flow, you can include one or more TryCatch nodes to provide a single point of failure for a sequence of nodes. With this technique, you can provide very specific error processing and recovery.

A TryCatch node does not process a message in any way, it represents only a decision point in a message flow. When the TryCatch node receives a message, it propagates it to the try terminal. The broker passes control to the sequence of nodes connected to that terminal (the try flow).

If an exception is thrown in the try flow, the broker returns control to the TryCatch node. The node writes the current contents of the ExceptionList to the local error log, then writes the information for the current exception to ExceptionList, overwriting the information stored there.

The node now propagates the message to the sequence of nodes connected to the catch terminal (the catch flow). The content of the message tree that is propagated is identical to the content that was propagated to the try terminal, which is the content of the tree when the TryCatch node first received it. It enhances this tree with the new exception information which it has written to ExceptionList. Any modifications or additions the nodes in try flow made to the message tree are not present in the message tree that is propagated to the catch flow.

However, if the try flow has completed processing that involves updates to external databases, these are not lost. The updates persist while the message is processed by the catch flow, and the decision about whether the updates are

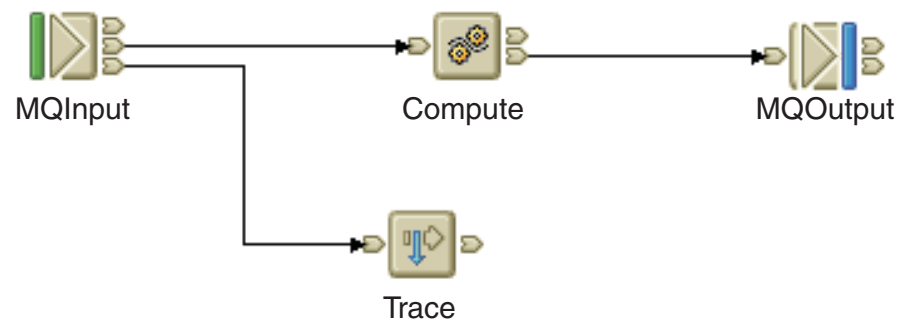
committed or rolled back is made on the configuration of your message flow and the individual nodes that interact with the databases. If the updates are committed because of the configuration you have set, you must include logic in your catch flow that rolls back the changes that were made.

To review the options for configuration, see “Configuring message flow transactions” on page 93.

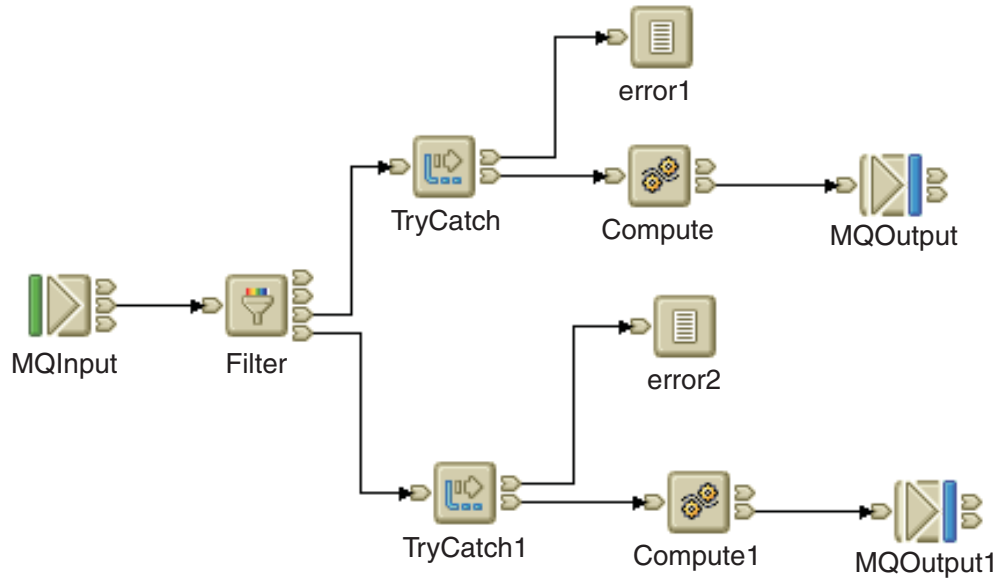
The broker returns control to the next catch point in the message flow (which might be another TryCatch node, but is always, in the last case, the input node) if:

- An exception is thrown in the catch flow of the TryCatch node (for example, if you include a Throw node, or code an ESQL THROW statement, or if the broker generates the exception).
- You do not connect the catch terminal of the TryCatch node.

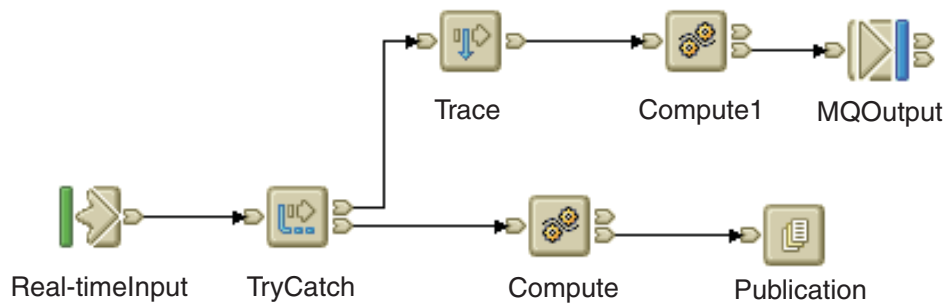
The example below shows how you can configure the flow to catch exceptions in the input node. The MQInput node’s catch terminal is connected to a Trace node to record the error.



In the example below, the message flow has two separate processing flows connected to the Filter node’s true and false terminals. Here a TryCatch node is included on each of the two routes that the message can take. The catch terminal of both TryCatch terminals is connected to a common error processing subflow.



If the input node in your message flow does not have a catch terminal (for example, Real-timeInput), and you want to process errors in the flow, you must include a TryCatch node. The example below shows how you could connect a flow to provide this error processing. In this example, you could configure the ESQL in the Compute node on the catch flow to examine the exception that has been caught and set the output queue name dynamically.



Managing message flows

This section contains information on managing message flows:

- “Creating a message flow project” on page 123
- “Deleting a message flow project” on page 124
- “Creating a broker schema” on page 125
- “Creating a message flow” on page 126
- “Opening an existing message flow” on page 127
- “Copying a message flow using copy” on page 127

- “Renaming a message flow” on page 128
- “Moving a message flow” on page 129
- “Deleting a message flow” on page 130
- Displaying version and keyword information
- “Saving a message flow” on page 132

To learn more about message flows, try importing the Airline Reservations sample (or another sample from the Samples Gallery) and explore the samples message flow resources; try creating, deleting, or renaming the resources.

Creating a message flow project

A message flow project is a container for message flows; you must create a project before you can create a message flow.

The project and its resources are stored in a file system or in a shared repository. If you are using a file system, this can be the local file system or a shared drive. If you store files in a repository, you can use any of the available repositories that are supported by Eclipse, for example CVS.

To create a message flow project:

1. Switch to the Broker Application Development perspective.
2. Click **File** → **New** → **Message Flow Project** or right-click any resource in the Navigator view and click **New** → **Message Flow Project**.
You can also press Ctrl+N. This displays a dialog that allows you to select the wizard to create a new object. Click Message Brokers in the left view; the right view displays a list of objects that you can create for WebSphere Message Broker. Click Message Flow Project in the right view, then click **Next**. The New Message Flow Project wizard displays.
3. Enter a name for the project. Choose a project name that reflects the message flows that it contains. For example, if you want to use this project for financial processing message flows, you might give it the name Finance_Flows.
4. Leave the *Use default* check box checked (it is checked when the dialog opens) This applies if you want to use the default location for the new message project directory, that is, in the \workspace subdirectory of your current installation. You cannot edit the *Directory* entry field.
 - a. Alternatively, clear the *Use default* check box and specify a location for the new message flow project files in the *Directory* entry field. This applies if you do not want to use the default location.
 - b. Use the **Browse** button to find the desired location or type the location in.
5. Click **Next** if you want to specify that this message flow project depends on other message flow projects, or on message set projects, You are presented with a list of current projects. Select one or more message flow projects, or one or more message set projects, or both, from the list to indicate this new message flow project’s dependencies.

This message flow project depends on another message flow project if you intend to use common resources within it. Common resources that you can share between message flow projects are:

- a. ESQ subroutines (defined in broker schemas)
- b. Mappings
- c. Message sets
- d. Subflows

For example, you might want to reuse a subflow that provides standard error processing such as writing the message to a database, or recording a trace entry.

This message flow project depends on a message set project if you intend to refer to the message it defines within ESQL within the message flow nodes.

You can add dependencies after you have created the message flow project by right-clicking the project in the Resource Navigator and clicking **Properties**.

Click **References** and select the dependent message flow or message set project from the list of projects displayed.

6. Click **Finish** to complete the task.

The project file is created within a directory that has the same name as your message flow project in the specified location. All other files that you create (or cause to be created) related to this message flow project are created in this same directory.

A default broker schema (default) is also created within the project. You can create and use different schemas within a single project to organize message flow resources, and to provide the scope of resource names to ensure uniqueness.

Deleting a message flow project

A message flow project is the container in which you create and maintain all the resources associated with one or more message flows. These resources are created as files, and are displayed within the project in the Resource Navigator view. If you do not want to retain a message flow project, you can delete it.

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow project” on page 123

Deleting a message flow project in the workbench deletes the project and its resources; the Configuration Manager does not hold a copy. If you are using a shared repository, the repository might retain a copy of a deleted resource.

In previous releases you could remove resources from the Control Center, which removed the reference in your workspace, but retained the resource in the Configuration Manager repository.

To delete a message flow project:

1. Switch to the Broker Application Development perspective.
2. Highlight the message flow project that you want to delete and click **Edit** → **Delete**. You can also press Del, or right-click the project in the Navigator view and click **Delete**.
3. You must choose if you want the contents of the message flow project folder deleted with this action on the displayed confirmation dialog. The dialog contains two buttons:
 - a. The first confirms that all contents are to be deleted.
 - b. The second requests that the directory contents are not deleted. The default action is not to delete the contents, and the second button is selected by default when the dialog is initially displayed.

- a. Select the appropriate button. If you choose not to delete the contents of the message flow project directory, all the files and the directory itself are retained.

If you later create another project with the same name, and specify the same location for the project (or accept this as the default value), you can access the files previously created.

If you choose to delete all the contents, all files and the directory itself are deleted.

4. Click **Yes** to complete the delete request, or **No** to terminate the delete request.

When you click **Yes**, the requested objects are deleted.

If you maintain resources in a shared repository, a copy is retained in that repository. You can follow the instructions provided by the repository supplier to retrieve the resource if required.

If you are using the local drive or a shared drive to store your resources, no copy of the resource is retained. Be very careful to select the correct resource when you complete this task.

Creating a broker schema

If you want to organize your message flow project resources, and to define the scope of resource names to ensure uniqueness, you can create broker schemas. A default schema is created when you create the message flow project, but you can create additional schemas if you choose.

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow project” on page 123

To create a broker schema:

1. Switch to the Broker Application Development perspective.
2. Click **File** → **New** → **BrokerSchema** or right-click any resource in the Navigator view and click **New** → **BrokerSchema**.

You can also press Ctrl+N. This displays a dialog that allows you to select the wizard to create a new object. Click Message Brokers in the left view. The right view displays a list of objects that you can create for WebSphere Message Broker. Click Broker Schema in the right view, then click **Next**. The New Broker Schema wizard displays.

3. Enter the message flow project in which you want the new schema to be created. If you have a message flow project or one of its resources highlighted when you invoke the wizard, that project name appears in the dialog. If a name does not appear in this field, or if you want to create the schema in another project, click **Browse** and select the correct project from the displayed list.

You can type the project name in, but you must enter a valid name. The dialog displays a red cross and the error message The specified project does not exist if your entry is not a valid project. You must specify a message flow project; if you select a message set project, the **Finish** button remains disabled.

4. Enter a name for the schema. Choose a name that reflects the resources that it contains. For example, if you want to use this schema for message flows for retail applications, you might give it the name Retail.

A broker schema name must be a character string that starts with a Unicode character followed by zero or more Unicode characters or digits, and the underscore. You can use the period to provide a structure to the name, for example `Stock.Common`.

5. Click **Finish** to complete the task.

The schema directory is created in the project directory. If the schema is structured using periods, further subdirectories are defined. For example, the broker schema `Stock.Common` results in a directory `Common` within a directory `Stock` within the message flow project directory.

Creating a message flow

Create a message flow to specify how to process messages in the broker. You can create any number of message flows and deploy them to one or more brokers.

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow project” on page 123

The message flow and its resources are stored in a file system or in a shared repository. If you are using a file system, this can be the local drive or a shared drive. If you store files in a repository, you can use any of the available repositories that are supported by Eclipse, for example CVS.

Use this process to create a complete message flow that you can deploy, or a subflow that provides a subset of function (for example, a reusable error processing routine) that you cannot deploy on its own.

To create a message flow:

1. Switch to the Broker Application Development perspective.
2. Check that you have already created the message flow project in which you want to create the message flow. You can only create a message flow in an existing project. The project can be empty, or can already have message flows defined in it.
3. Click **File** → **New** → **Message Flow** or right-click any resource in the Navigator view and click **New** → **Message Flow**.

You can also press **Ctrl+N**. This displays a dialog that allows you to select the wizard to create a new object. Click **Message Brokers** in the left view. The right view displays a list of objects that you can create for WebSphere Message Broker. Click **Message Flow** in the right view, then click **Next**. The New Message Flow wizard displays.

4. Identify the project in which you want to define the message flow. If you have a resource selected in the Navigator view, the name of the corresponding project is displayed in the first entry field, **Project**.

If you do not have a resource selected, the first field is blank. Click **Browse** to select the appropriate project for this message flow. A dialog containing a list of valid projects is displayed. Select the correct project and click **OK**.

You can type the project name in, but you must enter a valid name. The dialog displays a red cross and the error message `The specified project does not exist` if your entry is not a valid project.

5. Complete the **Schema** and **Name** fields when the project is correct:

- a. In **Schema**, enter the identifier of the broker schema in which the message flow is defined. When you create a message flow project, a default schema is created within it, and this default value is always assumed if you do not enter a value in this field, or do not select a value using the **Browse** button. You can create and use different schemas within a single project to organize message flow resources, and to provide the scope of resource names to ensure uniqueness.
 - b. In **Name**, enter the name of the message flow. You can use any valid character for the name; choose a name that reflects its function, for example `OrderProcessing`.
6. Click **Finish**.

The new message flow (`<message_flow_name>.msgflow`) is displayed within its project in the Navigator view. The editor view is empty and ready to receive your input.

Opening an existing message flow

Open an existing message flow to change or update its contents, or to add or remove nodes.

Before you start

To complete this task, you must have completed the following tasks:

- “Creating a message flow” on page 126

To open an existing message flow:

1. Switch to the Broker Application Development perspective. The Navigator view is populated with all the message flow and message set projects that you have access to. A message flow is contained in a file called `<message_flow_name>.msgflow`.
2. Right-click the message flow that you want to work with, and click **Open**. Alternatively you can double-click the message flow in the Navigator view. The graphical view of the message flow is displayed in the editor view. You can now work with this message flow, for example, adding or removing nodes, changing connections, or modifying properties.
3. Click **Open ESQL** for any node in the flow that requires ESQL, or you can double-click the ESQL file (the `.esql` file) in the Navigator view to open it, if you want to work with the ESQL file for this message flow.
4. Click **Open Mappings** for any node in the flow that requires mappings, or you can double-click the mappings file (the `.mfmap` file) in the Navigator view to open it, if you want to work with the mappings file for this message flow.

Copying a message flow using copy

You might find it useful to copy a message flow as a starting point for a new message flow that has similar function. For example, you might want to replace or remove one or two nodes to process messages in a different way.

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow” on page 126

To copy a message flow:

1. Switch to the Broker Application Development perspective.
2. Select the message flow (<message_flow_name>.msgflow) that you want to copy in the Navigator view.
 - a. Right-click the file and click **Copy** from the menu.
3. Right-click the broker schema within the message flow project to which you want to copy the message flow and click **Paste**. You can copy the message flow within the same broker schema within the same message flow, or to a different broker schema within the same message flow project, or to a broker schema in a different message flow project.

When you copy a message flow, the associated files (ESQL and mapping, if present) are not automatically copied to the same target message flow project. If you want these files copied as well, you must do this explicitly following this procedure.

You might also need to update nodes that have associated ESQL or mappings, to ensure that modules are unique.

For example, if you have created a message flow (Test1 for example) that contains a single Compute node, and you copy message flow Test1 and its associated .esql file to the same broker schema within the same message flow project (and give the new copy a different name, for example Test2), there are now two modules named Test1_Compute within the single schema. One is within Test1.esql, the second within Test2.esql.

This is not supported, and an error message is written to the Tasks view when you have completed the copy action. You must rename the associated ESQL modules within the .esql file and update the matching node properties to ensure that every module within a broker schema is unique.

The message flow is copied with all property settings intact. If you intend to use this copy of the message flow for another purpose, for example to retrieve messages from a different input queue, you might have to modify its properties.

You can also use **File** → **Save As** to copy a message flow. This is described in “Saving a message flow” on page 132.

Renaming a message flow

You can rename a message flow. You might want to do this if you have modified the message flow to provide a different function and you want the name of the message flow to reflect this new function.

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow” on page 126

To rename a message flow:

1. Switch to the Broker Application Development perspective.
2. Right-click the message flow that you want to rename (<message_flow_name>.msgflow) in the Navigator view, and click **Rename**, or click **File** → **Rename**. If you have the message flow selected, you can also press F2. The Rename Resource dialog is displayed.
3. Type in the new name for the message flow.

4. Click **OK** to complete this action, or **Cancel** to cancel the request. If you click **OK**, the message flow is renamed.
After you have renamed the message flow, any references that you have to this message flow (for example, if it is embedded in another message flow) are no longer valid.
5. You must open the affected message flows and correct the references if you are not sure where you have embedded this message flow.
 - a. Click **File** → **Save All**. The save action saves and validates all resources. Unresolved references are displayed in the Tasks view, and you can click each error listed.
This opens the message flow that makes a non-valid reference in the editor view
 - b. Right click the subflow icon and click **Locate Subflow**. The Locate Subflow dialog is displayed, listing the available message flow projects.
 - c. Expand the list and explore the resources available to locate the required subflow.
 - d. Select the correct subflow and click **OK**. All references in the current message flow are updated for you and the errors removed from the Tasks view.

Moving a message flow

You can move a message flow from one broker schema to another within the same project or to a broker schema in another project. You might want to do this, for example, if you are reorganizing the resources in your projects.



Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow” on page 126

To move a message flow:

1. Switch to the Broker Application Development perspective.
2. Drag and drop the message flow that you want to move from its current location to a broker schema within the same or another message flow project. If the target location that you have chosen is not valid, a black no-entry icon appears over the target, an error dialog is displayed, and the message flow is not moved.
You can move a message flow to another schema in the same project or to a schema in another message flow project.
3. If you prefer, you can:
 - a. Right-click the message flow that you want to move (<message_flow_name>.msgflow) in the Navigator view and click **Move**, or **File** → **Move**. The Move dialog is displayed. This contains a list of all valid projects to which you can move this message flow.
 - b. Select the project and the broker schema within the project to which you want to move the message flow. You can move a message flow to another schema in the same project or to a schema in another message flow project.
 - c. Click **OK** to complete the move, or **Cancel** to cancel the move. If you click **OK**, the message flow is moved to its new location.

4. Check the Tasks view for any errors (indicated by the error icon ) or warnings (indicated by the warning icon ) generated by the move. The errors in the Tasks view include those caused by broker references. When the move is done, all references to this message flow (for example, if this is a reusable error message flow that you have embedded in another message flow) are checked.
If you have moved the message flow within the same broker schema (in the same or another project), all references are still valid.
However, if you move the message flow from one broker schema to another (in the same or a different project), the references are broken.
This is because the resources are linked by a fully-qualified name of which the broker schema is a part. Information about any broken references is written to the Tasks view, for example, Linked or nested flow mflow1 cannot be located.
5. Double-click each error or warning to correct it. This opens the message flow that has the error in the editor view and highlights the node in error.

When you move a message flow, the associated files (for example, any ESQL or mapping files) are not automatically moved to the target broker schema. If you want these files moved as well, you must do this explicitly following this procedure.

Deleting a message flow

You can delete a message flow that you have created in a message flow project if you no longer need it.

Deleting a message flow in the workbench deletes the project and its resources, and the Configuration Manager does not hold a copy. If you are using a shared repository, the repository might retain a copy of a deleted resource.

In previous releases you could remove resources from the Control Center, which removed the reference in your workspace, but retained the resource in the Configuration Manager repository.

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow” on page 126

To delete a message flow:

1. Switch to the Broker Application Development perspective.
2. Select the message flow in the Navigator view (`<message_flow_name>.msgflow`) and press the Delete key. A confirmation dialog is displayed.

You can also right-click the message flow in the Navigator view and click **Delete**, or click **Edit** → **Delete**. The same dialog is displayed.

3. Click **Yes** to delete the message flow definition file or **No** to cancel the delete request. When you click **Yes**, the requested objects are deleted.

If you maintain resources in a shared repository, a copy is retained in that repository. You can follow the instructions provided by the repository supplier to retrieve the resource if required.

If you are using the local file system or a shared file system to store your resources, no copy of the resource is retained. Be very careful to select the correct resource when you complete this task.

4. Check the Tasks view for any errors that are caused by the deletion. Errors are generated if you delete a message flow that is embedded within another flow because the reference is no longer valid.
 - a. Click the error in the Tasks view This opens the message flow that now has a non-valid reference.
 - b. Either remove the node that represents the deleted message flow from the parent message flow, or create a new message flow with the same name to provide whatever processing is required.

When you delete the message flow, the files that are associated with the message flow (the ESQL and mapping files, if present) are not deleted by this action. If you want to delete these files also, you must do so explicitly.

Deleting a broker schema

You can delete a broker schema that you have created in a message flow project if you no longer need it.

Before you start

To complete this task, you must have completed the following task:

- “Creating a broker schema” on page 125

To delete a broker schema:

1. Switch to the Broker Application Development perspective.
2. Select the broker schema in the Navigator view and press the Delete key. A confirmation dialog is displayed.

You can also right-click the broker schema in the Navigator view and click **Delete**, or click **Edit** → **Delete**. The same dialog is displayed.

If the broker schema contains resources, the Delete menu option is disabled, and the Delete key has no effect. You must delete all resources within the schema before you can delete the schema.

3. Click **Yes** to delete the broker schema directory or **No** to cancel the delete request. When you click **Yes**, the requested objects are deleted.

If you maintain resources in a shared repository, a copy is retained in that repository. You can follow the instructions provided by the repository supplier to retrieve the resource if required.

If you are using the local file system or a shared file system to store your resources, no copy of the resource is retained. Be very careful to select the correct resource when you complete this task.

Object version and keyword

This topic contains information about how to view the version and keyword information of deployable objects.

- “Displaying object version in the bar file editor” on page 132
- “Displaying version, deploy time, and keywords of deployed objects” on page 132

Displaying object version in the bar file editor

A column in the bar editor called *Version* displays the version tag for all objects that have a defined version. These are:

- .dictionary files
- .cmf files
- Embedded JAR files with a version defined in a META-INF/keywords.txt file

You cannot edit the *Version* column.

You can use the `mqsireadbar` command to list the keywords defined for each deployable file within a deployable archive file.

Displaying version, deploy time, and keywords of deployed objects

The Eclipse *Properties View* displays, for any deployed object:

- Version
- Deploy Time
- All defined keywords

For example, if you deploy a message flow with these literal strings:

- `$MQSI_VERSION=v1.0 MQSI$`
- `$MQSI Author=fred MQSI$`
- `$MQSI Subflow 1 Version=v1.3.2 MQSI$`

then the Properties View displays:

Deployment Time	Date and time of deployment
Modification Time	Date and time of modification
Version	v1.0
Author	fred
Subflow 1 Version	v1.3.2

You are given a reason if the keyword information is not available. For example, if keyword resolution has not been enabled at deploy time, the Properties View displays the message Deployed with keyword search disabled. Also, if you deploy to a Configuration Manager that is an earlier version than Version 6.0, the properties view displays Keywords not available on this Configuration Manager.

Saving a message flow

You might want to save your message flow when you want to:

- Close the workbench.
- Work with another resource.
- Validate the contents of the message flow.

Before you start:


To complete this task, you must have completed the following task:

- “Creating a message flow” on page 126

To save a message flow:


1. Switch to the Broker Application Development perspective.
2. Select the editor view that contains the open message flow that you want to save.
3. If you want to save the message flow without closing it in the editor view, press Ctrl+S or click **File** → **Save name** on the taskbar menu (where *name* is the name of this message flow). You can also choose to save everything by clicking **File** → **Save All**.

The message flow is saved and the message flow validator is invoked to validate its contents. The validator provides a report of any errors that it finds in the Tasks view. The message flow remains open in the editor view.

For example, if you save a message flow and have not set a mandatory property, an error message appears in the Tasks view and the editor marks the node with the error icon . The message flow in the Navigator view is also marked with the error icon. This can occur if you have not edited the properties of an MQInput node to define the queue from which the input node retrieves its input messages.


(If you edit the properties of a node, you cannot click **OK** unless you have set all mandatory properties. Therefore this situation can arise only if you have never set any properties.)

You might also get warnings when you save a message flow. These are

indicated by the warning icon . This informs you that, although there is not an explicit error in the configuration of the message flow, there is a situation that might result in unexpected results when the message flow completes. For example, if you have included an input node in your message flow that you have not connected to any other node, you get a warning. In this situation, the editor marks the node with the warning icon. The message flow in the Navigator view is also marked with a warning icon.

4. If you save a message flow that includes a subflow, and the subflow is no longer available, three error messages are added to the Tasks view that indicate that the input and output terminals and the subflow itself cannot be located. This can occur if the subflow has been moved or renamed.

To resolve this situation, right-click the subflow node in error and click **Locate Subflow**. The Locate Subflow dialog is displayed, listing the available message flow projects. Expand the list and explore the resources available to locate the required subflow. Select the correct subflow and click **OK**. All references in the current message flow are updated for you and the errors removed from the Tasks view.

5. If you want to save the message flow when you close it, click the close view icon  on the editor view tab for this message flow or click **File** → **Close** on the taskbar menu. The editor view is closed and the file saved. The same validation occurs and any errors and warnings are written to the Tasks view.

For information about using the **File** → **Save As** option to take a copy of the current message flow, see “Copying a message flow using save” on page 134.

See “Correcting errors from the save action” on page 134 for information about handling errors from the save action.

Copying a message flow using save

You can copy a message flow by using the **File** → **Save As** option.

1. Click **File** → **Save name As**.
2. Specify the message flow project in which you want to save a copy of the message flow. The project name defaults to the current project. You can accept this name, or choose another name from the valid options that are displayed in the File Save dialog.
3. Specify the name for the new copy of the message flow. If you want to save this message flow in the same project, you must either give it another name, or confirm that you want to overwrite the current copy (that is, copy the flow to itself).

If you want to save this message flow in another project, the project must already exist (you can only select from the list of existing projects). You can save the flow with the same or another name in another project.

4. Click **OK**. The message flow is saved and the message flow editor validates its contents. The editor provides a report of any errors that it finds in the Tasks view. See “Correcting errors from the save action” for information about handling errors from the save action.

Correcting errors from the save action

Correct the errors that are reported when you save a message flow.

To correct errors from the save or save as action:

1. Examine the list of errors and warnings that the validator has generated in the Tasks view.
2. Double-click each entry in turn. The message flow is displayed in the editor view (if it is not already there), and the editor selects the node in which the error was detected. If the error has been generated because you have not set a mandatory property, the editor also opens the properties dialog for that node.
If you have included a user-defined node in your message flow, and have defined one or more of its properties as configurable, you might get a warning about a custom property editor. If you define a property as configurable, and you have specified that it uses a custom property editor, the bar editor cannot handle the custom property editor and handles the property as if it is type String. This restricts your ability to make changes to this property at deploy time.
3. Correct the error indicated by the message. For example, provide a value for the mandatory property.
4. When you have corrected all the errors, you can save again. The editor validates all the resources that you have changed, removes any corrected errors from the Tasks view, and removes the corresponding graphical indication from the nodes that you have modified successfully.

You do not have to correct every error to save your work. The editor saves your resources even if it detects errors or warnings, so that you can continue to work with them at a later date. However, you cannot deploy any resource that has a validation error. You must correct every error before you deploy a resource. Warnings do not prevent successful deployment.

Defining message flow content

When you create a new message flow, the editor view is initially empty. You must create the contents of the message flow by:

- “Adding a node”
- “Adding a subflow” on page 136
- “Renaming a node” on page 137
- “Configuring a node” on page 138
- “Connecting nodes” on page 140
- “Adding a bend point” on page 143
- “Aligning and arranging nodes” on page 144

When you finalize the content of the message flow, you might also need to perform the following tasks:

- “Removing a node” on page 139
- “Removing a connection” on page 142
- “Removing a bend point” on page 143

To learn more about message flow content, try importing the Airline Reservations sample or the Error Handler sample, and follow the supplied instructions to build the sample yourself. Also, try adding and deleting nodes, adding subflows, and connecting nodes together.

Adding a node

When you create a new message flow, the first action to take to define its function is to add nodes.

Before you start

To complete this task, you must have completed one of the following tasks:

- “Creating a message flow” on page 126
- “Opening an existing message flow” on page 127

To add a node to a message flow:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Click **Selection** above the palette of nodes.
4. Decide which node you want to add. This might be a built-in node or a user-defined node. You can select any of the nodes that appear in the node palette to the left of the editor view. You can only add one node at a time.
- 5.

In Message Brokers Toolkit, drag-and-drop the node that you want to include in the flow from the node palette into the editor view.

When you add a node into the editor view, the editor automatically assigns a name to the node. The name is equal to the type of node for the first instance. For example, if you add a Compute node to the editor view, it is assigned the name Compute. If you add a second instance, the default name assigned is appended by the character 1, for example, Compute1. The third is Compute2, and so on.

6. Repeat steps 4 on page 135 and 5 on page 135 to add further nodes.
7. You can also add nodes from other flows into this flow. To do this:
 - a. Open the other flow, select the node or nodes that you want to copy from the editor or outline views, and press **Ctrl+C** or click **Edit** → **Copy**.
 - b. Return to the flow that you are currently working with, and press **Ctrl+V** or click **Edit** → **Paste**. This action copies the node or nodes into your current flow. The node names and properties are preserved in the new copy.

When you have added the nodes that you want in this message flow, you can connect them to specify the flow of control through the message flow, and you can configure their properties.

Adding a node using the keyboard

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow” on page 126

You can use the Message Flow editor to perform tasks using the keyboard, such as adding a node to a message flow.

Complete the following steps to add a node to the canvas:

1. Open the message flow you want to add a node to by double-clicking the message flow in the Navigator view. You can also open the message flow by right-clicking it in the Navigator view and clicking **Open**. The message flow contents are displayed in the editor view.
2. Open the Palette view or the Palette bar.
3. Select a node in the Palette view or Palette bar using the up and down arrows to highlight the node you want to add to the canvas.
4. Add the Node to the canvas using one of the following methods:
 - Select **Palette** → **Add Node to Canvas** by pressing **Alt + L** and then pressing **N**.
 - Press **Shift + F10** to open the context-sensitive menu for the Palette, and Press **N**.

The node that you selected in the Palette bar or Palette view is placed on the canvas in the Editor view.

You can move the node that you have placed on the canvas using the keyboard controls described in Message Brokers Toolkit keyboard shortcuts

Adding a subflow

Within a message flow, you might want to include an embedded message flow, also known as a subflow. For example, you might define a subflow that provides error handling, and include it in a message flow connected to a failure terminal on a node that can generate an error in some situations.

Before you start

To complete this task, you must have completed one of the following tasks:

- “Creating a message flow” on page 126
- “Opening an existing message flow” on page 127

When you add a subflow, it appears in the editor view as a single node.

You can embed subflows into your message flow if either of the following statements is true:

- The flow that you want to embed is defined in the same message flow project.
- The flow is defined in a different message flow project, and you have specified the dependency of the current message flow project on that other project.

To add a subflow to a message flow:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Drag and drop the message flow from the Navigator view into the editor view. Alternatively, highlight the embedding message flow and click **Edit** → **Add subflow**, which displays a list of valid flows that you can add to the current flow.
4. Select the flow that you want to add from the list. The subflow icon is displayed with the terminals that represent the Input and Output nodes that you have included in the subflow.
5. Click **OK**.
6. Repeat steps 3, 4, and 5 to add further subflow nodes.
7. Select and open (double-click) the flow by name in the Navigator view, or right-click the embedded flow icon and select **Open Subflow** to work with the contents of the embedded flow

When you have added the nodes that you want in this message flow, you can connect them to specify the flow of control through the message flow, and you can modify their properties.

Renaming a node

You can change the name of a node. Its current name might be the default name that the editor assigns to it when you add the node to the editor view. Change the current name to reflect the purpose of the node. The node can be a built-in node, a user-defined node, or a subflow node.

For example, you might include a Compute node to calculate the price of a specific part within an order, and you could change the name of the node to be Calculate_Price.

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow” on page 126

When you rename a node, use only the supported characters for this entity. The editor prevents you from entering unsupported characters.

To rename a node:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Right-click the node that you want to rename in the editor view and click **Rename**. The Rename Node dialog is displayed.

4. Modify the existing name in the text entry field, or press the Delete key to erase the current name and enter a new one. The name that you enter must be unique within the message flow; you are prevented from entering a duplicate name.
5. Click **OK** to apply the new node name, or **Cancel** to end the dialog without changing the name.

Configuring a node

When you have included an instance of a node in your message flow, you can configure it to customize its function. The node can be a built-in node, a user-defined node, or a subflow node.

Before you start

To complete this task, you must have completed the following task:

- “Adding a node” on page 135

To configure a node:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Click **Selection** above the palette of nodes.
4. Right-click the node in the editor view, and click **Properties**. The node’s properties dialog is displayed.
5. You can select each group, displayed on the left as a tree navigation, in turn to view and modify the properties of the node. Every node has at least one group, Description. This contains two text entry fields, in which you can enter a short description, or a long description, or both. These are used only for documentation purposes, and are optional.

If the node only has Description properties (for example, the FlowOrder node), these are initially displayed when you open the properties dialog.

For most other nodes, the Basic properties are initially displayed. A few nodes do not have Basic properties; for these, the first group is displayed.

6. On the right are the properties of the currently-selected group. There can be one or more properties in this display. If the property is required, that is, one for which you must enter a value, the property name is marked with an asterisk.

For example, the Basic properties of the MQInput node include just one property, *Queue Name*. This identifies the queue from which input messages are retrieved by the node, and is a required property. On the properties dialog, it appears like this:

Queue Name* _____

- a. Make the changes that you want to make to the properties. In most cases, you cannot click **OK** to dismiss the properties dialog unless you have completed all mandatory properties.
7. When you have updated the properties of the node, click **OK** to save the changes, or **Cancel** to leave the dialog without saving your changes.

For details of how to configure each individual built-in node, see the node description. You can find a list of the nodes, with links to the individual topics, in “Built-in nodes” on page 482.

If you have included a user-defined node, refer to the documentation that came with the node to understand if, and how, you can configure its properties.

In the Message flow editor you can display node and connection metadata by hovering the mouse over a node or subflow in a message flow. To view metadata information for a node, subflow or connection:

1. Open the Broker Application Development Perspective
2. Open a message flow
3. In the Editor view, hover the mouse over a node, a subflow, or a node connection in the open message flow by placing the mouse over the element.

Promoted properties

You can promote node properties to their containing message flow. Use this technique to set some values at the message flow level, without having to change individual nodes. This can be useful, for example, when you embed a message flow in another flow, and want to override some property such as output queue or data source with a value that is correct in this context.

Overriding properties at deploy time

A small number of node property values can be overridden when a message flow is deployed. These are known as configurable properties, and you can use these to modify some characteristics of a deployed message flow without changing the message flow definitions. For example, you can update queue manager and data source information.

Even though you can set values for configurable properties at deploy time, you must set values for these properties within the message flow if they are mandatory.

Removing a node

When you have created and populated a message flow, you might need to remove a node to change the function of the flow, or to replace it with another more appropriate node. The node can be a built-in node, a user-defined node, or a subflow node.

Before you start

To complete this task, you must have completed one of the following tasks:

- “Adding a node” on page 135
- “Adding a subflow” on page 136

To remove a node:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Select the node in the editor view and press the Delete key.
4. Highlight the node and click **Edit** → **Delete**

You can also right-click the node in the editor view and click **Delete**, or right-click the node in the Outline view and click **Delete**. The editor removes the node. If you have created any connections between that node and any other node, those connections are also deleted when you delete the node.

5. If you delete a node in error, you can restore it by right-clicking in the editor view and clicking **Undo Delete**. The node and its connections, if any, are restored.
6.
You can also click **Edit** → **Undo Delete** or press Ctrl+Z.
7. If you undo the delete, but decide it is the correct delete action, you can right-click in the editor view and click **Redo Delete**.
You can also click **Edit** → **Redo Delete**.

Connecting nodes

When you include more than one node in your message flow, you must connect the nodes to indicate how the flow of control passes from input to output. The nodes can be built-in nodes, user-defined nodes, or subflow nodes.

Before you start

To complete this task, you must have completed one of the following tasks:

- “Adding a node” on page 135
- “Adding a subflow” on page 136

Your message flow might contain just one MQInput node, one Compute node, and one MQOutput node. Or it might involve a large number of nodes, and perhaps embedded message flows, that provide a number of paths that a message can travel through depending on its content. You might also have some error processing routines included in the flow. You might also need to control the order of processing.

You can connect a single output terminal of one node to the input terminal of more than one node (this is known as fan-out). If you do this, the same message is propagated to all target nodes, but you have no control over the order in which the subsequent paths through the message flow are executed (except with the FlowOrder node).

You can also connect the output terminal of several nodes to a single node input terminal (this is known as fan-in). Again, the messages that are received by the target node are not received in any guaranteed order.

When you have completed a connection, it is displayed as a black line, and is drawn as close as possible to a straight line between the connected terminals. This might result in the connection passing across other nodes. To avoid this, you can add bend points to the connection.

In the Message flow editor you can display node and connection metadata by hovering the mouse over a node or subflow in a message flow. To view metadata information for a node, subflow or connection:

1. Open the Broker Application Development Perspective
2. Open a message flow
3. In the Editor view, hover the mouse over a node, a subflow, or a node connection in the open message flow by placing the mouse over the element.

If you define a complex message flow, you might have to create a large number of connections. The principle is the same for every connection. You create connections

either by using the mouse, or by using the Terminal Selection dialog. See “Creating connections with the mouse” and “Creating connections with the Terminal Selection dialog” for more information.

Creating connections with the mouse

To connect one node to another using the mouse:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Click **Connection** above the node palette.
4. Click the terminal from which the connection is to be made, that is, the terminal from which the message is propagated from the current node. For example, you can click the failure terminal of the MQInput node, or the out terminal of the Compute node. You do not need to keep the mouse button pressed. If you are unsure which terminal is which, hover your mouse over each one; the terminal is highlighted in blue and a pop-up displays the name of the terminal.
5. Move your mouse and click again when the mouse pointer is above the in terminal of the next node in the message flow (to which the message now passes for further processing). The terminal is highlighted in blue and a popup displays the name of the terminal so that you can check that you have the correct terminal. The connection is made when you click on a valid in terminal. The connection appears as a black line between the two terminals.

In the Message flow editor you can display node and connection metadata by hovering the mouse over a node or subflow in a message flow. To view metadata information for a node, subflow or connection:

1. Open the Broker Application Development Perspective
2. Open a message flow
3. In the Editor view, hover the mouse over a node, a subflow, or a node connection in the open message flow by placing the mouse over the element.

Creating connections with the Terminal Selection dialog

You can use the Terminal Selection dialog to connect nodes. To connect one node to another in this way:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Click **Connection** above the node palette.
4. Click the node from which you want the connection to be made. The Terminal Selection dialog is displayed.
5. Select the terminal from the list of valid terminals on this node. Click **OK**. The dialog ends.
6. Click the node to which to make the connection. If there is only one in terminal on this node, the connection is made immediately. If there is more than one in terminal on this node, the Terminal Selection dialog is displayed again, this time listing the in terminals of the selected node. Select the correct terminal by clicking it, and click **OK**.

You can also make a connection as follows:

1. Click **Selection** above the node palette.

2. Right-click the node from which you want to make the connection and click **Create Connection**. The Terminal Selection dialog is displayed.
3. Select the terminal from the list of valid terminals on this node. Click **OK**. The dialog ends.
4. Click the node to which to make the connection. If there is only one in terminal on this node, the connection is made immediately. If there is more than one in terminal on this node, the Terminal Selection dialog is displayed again, this time listing the in terminals of the selected node. Highlight the correct terminal and click **OK**.

In the Message flow editor you can display node and connection metadata by hovering the mouse over a node or subflow in a message flow. To view metadata information for a node, subflow or connection:

1. Open the Broker Application Development Perspective
2. Open a message flow
3. In the Editor view, hover the mouse over a node, a subflow, or a node connection in the open message flow by placing the mouse over the element.

Removing a connection

The message flow editor displays the nodes and connections in the editor view. You can remove connections to change the way in which the message flow processes messages.

Before you start

To complete this task, you must have completed the following task:

- “Connecting nodes” on page 140

If you want to remove a connection that you have created between two nodes:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Click **Selection** above the node palette.
4. Select the connection that you want to delete. When you hover your mouse pointer over the connection, the editor highlights the connection that you have selected by thickening its line, adding an arrowhead at the target terminal end, and annotating the connection with the name of the two terminals connected, for example Out->In.

When you select the connection, the editor appends a small black square at each end and at every bend point of the connection, and a small arrowhead at the target terminal end. The annotation disappears when you select the connection.

5. Check that the selected connection is the one that you want to delete.
6. Right-click the connection and click **Delete**, press the Delete key, or click **Edit** → **Delete**. If you want to delete further connections, repeat these actions from step 4.
7. If you delete a connection in error, you can restore it by right-clicking in the editor view and clicking **Undo Delete**. The connection is restored.
8. If you undo the delete, but decide that it is the correct delete action, you can right-click in the editor view and click **Redo Delete**. You can also delete a connection by selecting it in the Outline view and pressing the Delete key.

If you delete a node, its connections are automatically removed; you do not have to do this as a separate task.

Adding a bend point

When you are working with a message flow, and connecting your chosen nodes together to determine the flow of control, you might find that a connection that you have made crosses over an intervening node and makes the flow of control difficult to follow.

To help you to display the message flow nodes and their connections in a clear way, you can add bend points to the connections that you have made to improve the organization of the display. The addition of bend points has no effect on the execution of the nodes or the operation of the message flow.

Before you start

To complete this task, you must have completed the following task:

- “Connecting nodes” on page 140

To add a bend point:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Click **Selection** above the node palette.
4. Select the connection to which you want to add a bend point. The editor appends a small black square to each end of the connection to highlight it.
 - a. Check that this is the correct connection. The editor also adds a small point (a handle) in the connection halfway between the in and out terminals that are joined by this connection.
5. Hover your mouse pointer over this point until the editor displays a black cross to indicate that you now have control of this bend point.
 - a. Hold down the left mouse button and move your mouse to move the black cross and bend point across the editor view.
6. As you drag your mouse, the connection is updated, retaining its start and end points with a bend point at the drag point. You can move this anywhere within the editor view to improve the layout of your message flow.
7. Release the mouse button when the connection is in the correct place. The editor now displays the bend point that you have created with a small square (like those at the ends of the connection), and displays another two small points within the connection, one between your newly-created bend point and the out terminal, the other between the new bend point and the in terminal.

If you want to add more than one bend point to the same connection, repeat these actions from step 4 using the additional small points inserted into the connection.

Removing a bend point

When you are working with a message flow in the editor view, you might want to simplify the display of the message flow by removing a bend point that you previously added to a connection between two nodes.

Before you start

To complete this task, you must have completed the following task:

- “Adding a bend point” on page 143

To remove a bend point:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Click **Selection** above the node palette.
4. Select the connection from which you want to remove the bend point. The editor highlights the connection and its current bend points by thickening its line and appending a small black square to each end of the connection, and by indicating each bend point with a small black square. Check that this is the correct connection.
5. Right-click over the selected connection, if you added this bend point in the current edit session.
 - a. Click **Undo Create Bend Point**.

The editor removes the selected bend point.

If you right-click in the editor view without a connection being selected, you can also click **Undo Create Bend Point** from the menu. However, this removes the last bend point that you created in any connection, which might not be the one that you want to remove.

6. Move the bend point to straighten the line if you added this bend point in a previous edit session, because you cannot use the undo action. When the line is straight, the bend point is removed automatically.

When the bend point has been removed, the connection remains highlighted. Both ends of the connection, and any remaining bend points, remain displayed as small black squares. The editor also inserts small points (handles) into the connection between each bend point and between each terminal and its adjacent bend point, which you can use to add more bend points if you choose.

7. If you want to remove another bend point from the same connection, repeat these actions from step 4.

Aligning and arranging nodes

When you are working in the Message Flow editor, you can decide how your nodes are aligned within the editor view.

This option is closely linked to the way in which your nodes are arranged. Again, the default for this is left to right, which means that the in terminal of a node appears on its left edge, and its out terminals appear on its right edge. You can also change this characteristic of a node by rotating the icon display to right to left, top to bottom, and bottom to top.

Before you start

To complete this task, you must have completed the following task:

- “Adding a node” on page 135

To modify the way in which nodes and connections are displayed in the editor:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Click **Selection** above the node palette.

4. Right-click in the editor window and select **Manhattan Layout** if you want the connections between the nodes to be displayed in Manhattan style; that is with horizontal and vertical lines joined at right angles.
5. If you want to change the layout of the complete message flow:
 - a. Right-click in the editor view and click **Layout**. The default for the alignment is left to right, such that your message flow starts (with an input node) on the left and control passes to the right.
 - b. From the four further options displayed, **Left to Right**, **Right to Left**, **Top to Bottom**, and **Bottom to Top**, click the option that you want for this message flow. The message flow display is updated to reflect your choice. As a result of the change in alignment, all the nodes within the message flow are also realigned.
 For example, if you have changed from a left to right display (the default) to a right to left display, each node in the flow has now also changed to right to left (that is, the in terminal now appears on the right edge, the out terminals appear on the left edge).
6. You might want to arrange an individual node in a different direction from that in which the remaining nodes are arranged within the message flow, To do this:
 - a. Right-click the node that you want to change and click **Rotate**. This gives you four further options: **Left to Right**, **Right to Left**, **Top to Bottom**, and **Bottom to Top**.
 - b. Click the option that you want for this node. The option that represents the current arrangement of the node is not available for selection.

If you change the alignment of the message flow, or the arrangement of an individual node, or both, these settings are saved when you save the message flow. They are applied when another user accesses this same message flow, either through a shared repository or through shared files or import and export. When you reopen the message flow, you see these changed characteristics. The alignment and arrangement that you have selected for this message flow have no impact on the alignment and arrangement of any other message flow.

In the Message Brokers Toolkit Version 5.1 you can adjust the zoom by right-clicking in the editor view and clicking **Zoom in** or **Zoom out**. Alternatively, you can use the drop-down list on the editor toolbar to specify a zoom percentage.

You can also access the editor toolbar to select other options related to the display and arrangement of nodes, for example, snap to grid. These are defined in Message Flow editor.

Developing ESQL

When you use the built-in nodes Compute, Database, and Filter, you must customize them to determine the exact processing that they provide. To do this, you must create, for each node, an ESQL module in which you code the ESQL statements and functions to tailor the behavior of the node, referring to message content, or database content, or both, to achieve the results that you require. ESQL modules are maintained in ESQL files, managed through the Broker Application Development perspective.

This section provides information on:

- “ESQL overview” on page 146
- “Managing ESQL files” on page 156
- “Writing ESQL” on page 168

You can use the ESQL debugger, which is part of the flow debugger, to debug the code that you write. The debugger steps through ESQL code statement by statement, so that you can view and check the results of every line of code that is executed.

Note: In previous releases there were several types of debugger, each of which handled a specific type of code, such as ESQL, message flows, or Java. In Version 6, these separate debuggers are integrated into a single debugger, which is known simply as “the debugger”, and which handles all types of code.

ESQL overview

Extended Structured Query Language (ESQL) is a programming language defined by WebSphere Message Broker to define and manipulate data within a message flow.

This section contains introductory information about ESQL.

- For descriptions of ESQL user tasks, see “Writing ESQL” on page 168.
- For reference information about ESQL, see “ESQL reference” on page 779.

You are strongly recommended to read the following information before you proceed:

- An overview of message flows, see “Message flows overview” on page 3.
- An overview of message trees, see “The message tree” on page 9, and the topics within this container, paying special attention to “Logical tree structure” on page 14.

ESQL is based on Structured Query Language (SQL) which is in common usage with relational databases such as DB2. ESQL extends the constructs of the SQL language to provide support for you to work with message and database content to define the behavior of nodes in a message flow.

The ESQL code that you create to customize nodes within a message flow is defined in an ESQL file, typically named <message_flow_name>.esql,, which is associated with the message flow project. You can use ESQL in the following built-in nodes:

- “Compute node” on page 492
- “Database node” on page 500
- “Filter node” on page 515

You can also use ESQL to create functions and procedures that you can use in the following built-in nodes:

- “DataDelete node” on page 504
- “DataInsert node” on page 507
- “DataUpdate node” on page 510
- “Extract node” on page 513
- “Mapping node” on page 567
- “Warehouse node” on page 661

To use ESQL correctly and efficiently in your message flows, you must also understand the following concepts:

- Data types
- Variables
- Field references

- Operators
- Statements
- Functions
- Procedures
- Modules

Use the ESQL debugger, which is part of the flow debugger, to debug the code that you write. The debugger steps through ESQL code statement by statement, so that you can view and check the results of every line of code that is executed.

Note: In previous releases there were several types of debugger, each of which handled a specific type of code, such as ESQL, message flows, or Java. In Version 6, these separate debuggers are integrated into a single debugger, which is known simply as “the debugger”, and which handles all types of code.

ESQL data types

A data type defines the characteristics of an item of data, and determines how that data is processed. ESQL supports six data types, listed below. Data that is retrieved from databases, received in a self-defining message, or defined in a message model (using MRM data types), is mapped to one of these basic ESQL types when it is processed in ESQL expressions.

Within a broker, the fields of a message contain data that has a definite data type. It is also possible to use intermediate variables to help process a message. You must declare all such variables with a data type before use. A variable’s data type is fixed; If you try to assign values of a different type you get either an implicit cast or an exception. Message fields do not have a fixed data type, and you can assign values of a different type. The field adopts the new value and type.

It is not always possible to predict the data type that results from evaluating an expression. This is because expressions are compiled without reference to any kind of message schema, and so some type errors are not caught until runtime.

ESQL defines the following categories of data. Each category contains one or more data types.

- Boolean
- Datetime
- Null
- Numeric
- Reference
- String

ESQL variables

An ESQL variable is a data field used to help process a message.

You must declare a variable and state its type before you can use it. A variable’s data type is fixed; if you code ESQL that assigns a value of a different type, either an implicit cast to the data type of the target is implemented or an exception is raised (if the implicit cast is not supported).

To define a variable and give it a name, use the DECLARE statement.

Note: The names of ESQL variables are case sensitive, so it is important to make sure that you use the correct case in all places. The simplest way to guarantee this is always to define variables using upper case names.

The Message Broker Toolkit flags, with warning markers, variables that have not been defined. It is best practice to remove all these warnings before deploying a message flow.

You can assign an initial value to the variable on the DECLARE statement. If an initial value isn't specified, scalar variables are initialized with the special value NULL, while ROW variables are initialized to an empty state. Subsequently, you can change the variable's value using the SET statement.

There are three types of built-in node that can contain ESQL code and hence support the use of ESQL variables:

- "Compute node" on page 492
- "Database node" on page 500
- "Filter node" on page 515

Variable scope, lifetime, and sharing

How widespread and for how long a particular ESQL variable is available, is described by its scope, lifetime, and sharing:

A **variable's scope** is a measure of the range over which it is visible. In the broker environment, the scope of variables is normally limited to the individual node.

A **variable's lifetime** is a measure of the time for which it retains its value. In the broker environment, the lifetime of a variable varies but is typically restricted to the life of a thread within a node.

A **variable's sharing characteristics** indicate whether each thread has its own copy of the variable or whether one variable is shared between many threads. In the broker environment, variables are typically not shared.

Types of variable

You can use the "DECLARE statement" on page 851 to define three types of variable:

External

External variables (defined with the EXTERNAL keyword) are also known as *user-defined properties* (UDPs): see "User-defined properties in ESQL" on page 149. They exist for the entire lifetime of a message flow and are visible to all messages passing through the flow. Their initial values (optionally set by the DECLARE statement) can be modified, at design time, by the Message Flow editor, or, at deployment time, by the BAR editor. Their values cannot be modified by ESQL.

Normal

"Normal" variables have a lifetime of just one message passing through a node. They are visible to that message only. To define a "normal" variable, omit both the EXTERNAL and SHARED keywords.

Shared

Shared variables can be used to implement an in-memory cache in the message flow, see "Optimizing message flow response times" on page 73. Shared variables have a long lifetime and are visible to multiple messages passing through a flow, see "Long-lived variables" on page 150. They exist for the lifetime of the execution group process, the lifetime of the flow or node, or the

lifetime of the node's SQL that declares the variable (whichever is the shortest). They are initialized when the first message passes through the flow or node after each broker start up.

See also the ATOMIC option of the "BEGIN ... END statement" on page 807. The BEGIN ATOMIC construct is useful when a number of changes need to be made to a shared variable and it is important to prevent other instances seeing the intermediate states of the data.

For information about specific types of variable, see:

- "User-defined properties in ESQL" (external variables)
- "Long-lived variables" on page 150 (shared variables)

User-defined properties in ESQL: A user-defined property (UDP) is a variable that is defined in your ESQL program by specifying the EXTERNAL keyword on a "DECLARE statement" on page 851. For example, the ESQL statement DECLARE today EXTERNAL CHARACTER 'monday' defines a user-defined property called today with an initial value 'monday'.

Before you can use a user-defined property, you must also define the property when you construct a message flow that uses it. Use the Message Flow editor to do this.

When you define a UDP using the Message Flow editor, a value and property type is also defined. The value might be a default value, which varies according to the UDP's type. The value assigned to the UDP in the Message Flow editor takes precedence over any value that you have assigned to the UDP in your ESQL program.

Before you deploy the message flow that uses the UDP, you can change the value of the UDP by using the Broker Archive editor. A deployment failure occurs if you try to deploy a message flow that contains a UDP that has had no value assigned to it.

See "Configuring a message flow at deployment time using UDPs" on page 284 for more information.

Using UDPs, configuration data can be set easily and used just like normal constants. Because no external calls to user-written plug-ins or parsing of environment trees are involved, the ESQL code is easier to write and maintain, and performs better. Also, the parsing costs of reading data out of trees are removed. The value of the UDP is stamped into the variable at deployment time, which makes it quick to access.

You can declare UDPs only in modules or schemas.

UDPs can be accessed by any built-in node that uses ESQL:

- Compute
- Database
- Filter
- Nodes derived from these node-types, for example DataInsert, DataDelete, and DataUpdate.

See "Accessing user-defined properties from a JavaCompute node" on page 300 for a description of how to access a UDP from a JavaCompute node.

Long-lived variables: It is sometimes desirable to store data for longer than the lifetime of a single message passing through a flow. One way to do this, is to store the data in a database. This is good for long-term persistence and transactionality, but access (particularly write access) is slow.

Alternatively, you can use appropriate “long-lived” ESQL data types to provide an in-memory cache of the data for a certain period of time. This makes access much faster than it would be from a database, though this is at the expense of shorter persistence and no transactionality.

Long-lifetime variables are created by using the SHARED keyword on the DECLARE statement.

The Message Routing sample demonstrates how to define shared variables using the DECLARE statement. The sample demonstrates how to store routing information in a database table and use shared variables to store the database table in memory in the message flow to improve performance.

Long-lived data types have an extended lifetime beyond that of a single message passing through a node. They are shared between threads and exist for the life of a message flow (strictly speaking the time between configuration changes to a message flow), as described in this table:

	Scope	Life	Shared
Short lifetime variables			
Schema & Module	Node	Thread within node	Not at all
Routine Local	Node	Thread within routine	Not at all
Block Local	Node	Thread within block	Not at all
Long lifetime variables			
Node Shared	Node	Life of node	All threads of flow
Flow Shared	Flow	Life of flow	All threads of flow

Features of long-lived ESQL data types include:

- Ability to handle large amounts of long-lifetime data.
- Joining data to messages is fast.
- On multiple processor machines, multiple threads are able to access the same data simultaneously.
- Subsequent messages can access the data left by a previous message.
- Long lifetime read-write data can be shared between threads, because there is no long term association between threads and messages.
- In contrast to data stored in database tables in the “environment”, this sort of data is stored “privately”; that is, within the broker.
- The use of ROW variables can be used to create a modifiable copy of the input message.
- It is possible to create shared constants.

A typical use of these data types might be in a flow in which data tables are 'read-only' as far as the flow is concerned. Although the table data is not actually static, the flow does not change it, and thousands of messages pass through the flow before there is any change to the table data.

An example is a table which contains a day's credit card transactions. The table is created each day and that day's messages will be run against it. Then the flow is stopped, the table updated and the next day's messages run. It is very likely that such flows would perform better if they cached the table data rather than read it from a database for each message.

Another use of these data types might be the accumulation and integration of data from multiple messages.

Broker properties

For each broker, WebSphere Message Broker maintains a set of properties. You can access some of these properties from your ESQL programs. A subset of the properties is also accessible from Java code. It can be useful, during the runtime of your code, to have real-time access to details of a specific node, flow, or broker.

There are four categories of broker properties:

- Those relating to a specific node
- Those relating to nodes in general
- Those relating to a message flow
- Those relating to the execution group

"Broker properties accessible from ESQL and Java" on page 983 shows the broker, flow, and node properties that are accessible from ESQL and indicates which properties are also accessible from Java.

Broker properties:

- Are grouped by broker, execution group, flow, and node.
- Are case sensitive. Their names always start with an uppercase letter.
- Return NULL if they do not contain a value.

All nodes that allow user programs to edit ESQL support access to broker properties. These are:

- Compute nodes
- Database nodes
- Filter nodes
- All derivatives of these nodes

ESQL field references

An ESQL field reference is a sequence of period-separated values that identify a specific field (which might be a structure) within a message tree or a database table. The path from the root of the information to the specific field is traced using the parent/child relationships.

A field reference is used in an ESQL statement to identify the field that is to be referenced, updated, or created within the message or database table. For example, you might use the following identifier as a message field reference:

Body.Invoice.Payment

You can use an ESQL variable of type REFERENCE to set up a dynamic pointer to contain a field reference. This might be useful in creating a fixed reference to a commonly-referenced point within a message; for example the start of a particular structure that contains repeating fields.

A field reference can also specify element types, XML namespace identifications, indexes and a type constraint. These are discussed in detail later.

The first name in a field reference is sometimes known as a *Correlation name*.

ESQL operators

An ESQL operator is a character or symbol that you can use in expressions to specify relationships between fields or values.

ESQL supports the following groups of operators:

- Comparison operators, to compare one value to another value (for example, less than). Refer to “ESQL simple comparison operators” on page 798 for details of the supported operators and their use.
- Logical operators, to perform logical operations on one or two terms (for example, AND). Refer to “ESQL logical operators” on page 802 for details of the supported operators and their use.
- Numeric operators, to indicate operations on numeric data (for example, +). Refer to “ESQL numeric operators” on page 803 for details of the supported operators and their use.

There are some restrictions on the application of some operators to data types; not all lead to a meaningful operation. These are documented where they apply to each operator.

Operators that return a boolean value (TRUE or FALSE), for example the greater than operator, are also known as predicates.

ESQL statements

An ESQL statement is an instruction that represents a step in a sequence of actions or a set of declarations.

ESQL provides a large number of different statements that perform different types of operation. All ESQL statements start with a keyword that identifies the type of statement and end with a semicolon. An ESQL program consists of a number of statements that are processed in the order they are written.

As an example, consider the following ESQL program:

```
DECLARE x INTEGER;  
SET x = 42;
```

This program consists of two statements. The first starts with the keyword DECLARE and ends at the first semicolon. The second statement starts with the keyword SET and ends at the second semicolon. These two statements are written on separate lines and it is conventional (but not required) that they be so. You will notice that the language keywords are written in capital letters. This is also the convention but is not required; mixed and lower case are acceptable.

The first statement declares a variable called x of type INTEGER, that is, it reserves a space in the computer’s memory large enough to hold an integer value and allows this space to be subsequently referred to in the program by the name x. The

second statement sets the value of the variable x to 42. A number appearing in an ESQL program without decimal point and not within quotes is known as an integer literal.

ESQL has a number of data types and each has its own way of writing literal values. These are described in “ESQL data types” on page 147.

For a full description of all the ESQL statements, see “ESQL statements” on page 804.

ESQL nested statements: An ESQL nested statement is a statement that is contained within another statement.

Consider the following ESQL program fragment:

```
IF Size > 100.00 THEN
  SET X = 0;
  SET Y = 0;
  SET REVERSE = FALSE;
ELSE
  SET X = 639;
  SET Y = 479;
  SET REVERSE = TRUE;
END IF;
```

In this example, you can see a single IF statement containing the optional ELSE clause. Both the IF and ELSE portions contain three nested statements. Those within the IF clause are executed if the operator > (greater than) returns the value TRUE (that is, if Size has a value greater than 100.00); otherwise, those within the ELSE clause are processed.

Many statements can have expressions nested within them, but only a few can have statements nested within them. The key difference between an expression and a statement is that an expression calculates a value to be used, whereas a statement performs an action (usually changing the state of the program) but does not produce a value.

ESQL functions

A function is an ESQL construct that calculates a value from a number of given input values.

A function usually has input parameters and can, but does not usually have, output parameters. It returns a value calculated by the algorithm described by its statement. This statement is usually a compound statement, such as BEGIN... END, because this allows an unlimited number of nested statements to be used to implement the algorithm.

ESQL provides a number of predefined, or “built-in”, functions which you can use freely within expressions. You can also use the CREATE FUNCTION statement to define your own functions.

When you define a function, you must give it a unique name. The name is handled in a case insensitive way (that is, use of the name with any combination of upper and lower case letters matches the declaration). This is in contrast to the names that you declare for schemas, constants, variables, and labels, which are handled in a case sensitive way, and which you must specify exactly as you declared them.

Consider the following ESQL program fragment:

```
SET Diameter = SQRT(Area / 3.142) * 2;
```

In this example, the function SQRT (square root) is given the value inside the brackets (itself the result of an expression, a divide operation) and its result is used in a further expression, a multiply operation. Its return value is assigned to the variable Diameter. See “Calling ESQL functions” on page 892 for information about all the built-in ESQL functions.

In addition, an ESQL expression can refer to a function in another broker schema (that is, a function defined by a CREATE FUNCTION statement in an ESQL file in the same or in a different dependent project). To resolve the name of the called function, you must do one of the following:

- Specify the fully-qualified name (<SchemaName>.<FunctionName>) of the called function.
- Include a PATH statement to make all functions from the named schema visible. Note that this technique only works if the schemas do not contain identically-named functions. The PATH statement must be coded in the same ESQL file, but not within any MODULE.

Note that you cannot define a function within an EVAL statement or an EVAL function.

ESQL procedures

An procedure is a subroutine that has no return value. It can accept input parameters from, and return output parameters to, the caller.

Procedures are very similar to functions. The main difference between them is that, unlike functions, procedures have no return value. Thus they cannot form part of an expression and are invoked by using the CALL statement. Procedures commonly have output parameters

You can implement a procedure in ESQL (an internal procedure) or as a database stored procedure (an external procedure). The ESQL procedure must be a single ESQL statement, although that statement can be a compound statement such as BEGIN END. You cannot define a procedure within an EVAL statement or an EVAL function.

When you define a procedure, give it a name. The name is handled in a case insensitive way (that is, use of the name with any combination of upper and lower case letters matches the declaration). That is in contrast to the names that you declare for schemas, constants, variables, and labels, which are handled in a case sensitive way, and which you must specify exactly as you declared them.

An ESQL expression can include a reference to a procedure in another broker schema (defined in an ESQL file in the same or a different dependent project). If you want to use this technique, either fully qualify the procedure, or include a PATH statement that sets the qualifier. The PATH statement must be coded in the same ESQL file, but not within a MODULE.

An external database procedure is indicated by the keyword EXTERNAL and the external procedure name. This procedure must be defined in the database and in the broker, and the name specified with the EXTERNAL keyword and the name of

the stored database procedure must be the same, although parameter names do not have to match. The ESQL procedure name can be different to the external name it defines.

Overloaded procedures are not supported to any database. (An overloaded procedure is one that has the same name as another procedure in the same database schema which has a different number of parameters, or parameters with different types.) If the broker detects that a procedure has been overloaded, it raises an exception.

Dynamic schema name resolution for stored procedures is supported; when you define the procedure you must specify a wildcard for the schema that is resolved before invocation of the procedure by ESQL. This is explained further in “Invoking stored procedures” on page 212.

ESQL modules

A module is a sequence of declarations that define variables and their initialization, and a sequence of subroutine (function and procedure) declarations that define a specific behavior for a message flow node.

A module must begin with the CREATE node_type MODULE statement and end with an END MODULE statement. The node_type must be one of COMPUTE, DATABASE, or FILTER. The entry point of the ESQL code is the function named MAIN, which has MODULE scope.

Each module is identified by a name which follows CREATE node_type MODULE. The name might be created for you with a default value, which you can modify, or you can create it yourself. The name is handled in a case insensitive way (that is, use of the name with any combination of upper and lower case letters matches the declaration). That is in contrast to the names that you declare for schemas, constants, variables, and labels, which are handled in a case sensitive way, and which you must specify exactly as you declared them.

You must create the code for a module in an ESQL file which has a suffix of .esql. You must create this file in the same broker schema as the node that references it. There must be one module of the correct type for each corresponding node, and it is specific to that node and cannot be used by any other node.

When you create an ESQL file (or complete a task that creates one), you indicate the message flow project and broker schema with which the file is associated as well as specifying the name for the file.

Within the ESQL file, the name of each module is determined by the value of the corresponding property of the message flow node. For example, the property *ESQL Module* for the Compute node specifies the name of the node’s module in the ESQL file. The default value for this property is the name of the node. You can specify a different name, but you must ensure that the value of the property and the name of the module that provides the required function are the same.

The module must contain the function MAIN, which is the entry point for the module. This is included automatically if the module is created for you. Within MAIN, you can code ESQL to configure the behavior of the node. If you include ESQL within the module that declares variables, constants, functions, and procedures, these are of local scope only and can be used within this single module.

If you want to reuse ESQL constants, functions, or procedures, you must declare them at broker schema level. You can then refer to these from any resource within that broker schema, in the same or another project. If you want to use this technique, either fully qualify the procedure, or include a PATH statement that sets the qualifier. The PATH statement must be coded in the same ESQL file, but not within any MODULE.

Managing ESQL files

Within a message flow project, you can create ESQL files to contain the ESQL code that you provide to modify or customize the behavior of Compute, Database, or Filter nodes.

The ESQL code is contained within a module that is associated with the node. Each module must be created within an ESQL file. The name of the module within the ESQL file must match the name specified for the module in the *ESQL Module* property of the corresponding node. Although you can modify the module name, and change it from its default value (which is the name of the message flow, concatenated with the name of the node with which the module is associated), ensure that the module in the ESQL file matches the node property.

The following topics describe how you can manage these files:

- “Creating an ESQL file”
- “Opening an existing ESQL file” on page 158
- “Creating ESQL for a node” on page 158
- “Modifying ESQL for a node” on page 161
- “Saving an ESQL file” on page 162
- “Copying an ESQL file” on page 164
- “Renaming an ESQL file” on page 164
- “Moving an ESQL file” on page 165
- “Changing ESQL preferences” on page 166
- “Deleting ESQL for a node” on page 167
- “Deleting an ESQL file” on page 168

Creating an ESQL file

When you include a node in your message flow that requires ESQL to customize its function (the Compute, Database, and Filter nodes), you must code the ESQL statements that provide the customization in an ESQL module within an ESQL file. You can use the same ESQL file for more than one module, if you choose.

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow project” on page 123

ESQL files are stored in a file system or in a shared repository. If you are using a file system, this can be the local file system or a shared drive. If you store files in a repository, you can use any of the available repositories that are supported by Eclipse, for example CVS.

To create an ESQL file:

1. Switch to the Broker Application Development perspective.
2. Click **File** → **New** → **Message Flow ESQL File**.

You can also press Ctrl+N. This displays a dialog that allows you to select the wizard to create a new object. Click Message Brokers in the left view; the right view displays a list of objects that you can create for WebSphere Message Broker. Click Message Flow ESQL File in the right view, then click **Next**. The New Message Flow ESQL File wizard is displayed.

3. Enter the name of the message flow project in which to create the ESQL file. You must enter the name of an existing message flow project. The dialog is displayed with the current project name entered in the project name field. You can accept this value or change it to specify a different project. You can also click **Browse** to view a list of valid projects (projects that are defined and displayed in the Navigator view), and select the appropriate value from that list.

If you type in the name of a project that does not exist, the error message The specified project does not exist is displayed in the dialog and you cannot continue until you specify a valid project name.

4. If you want the ESQL file to be defined within a specific broker schema, enter the name of the broker schema in the appropriate entry field, or click **Browse** to select the broker schema from the list of valid broker schema for this project. (If only the default broker schema is defined in this project, **Browse** is disabled.)
5. Enter a name for the new ESQL file. If you enter a name that is already in use for an ESQL file in this project, the error message The resource <name>.esql already exists is displayed in the dialog and you cannot continue until you specify a valid name.

When creating ESQL files, the overall file path length must not exceed 256 characters, due to a Windows file system limitation. If you try to add a message flow to a broker archive file with ESQL or mapping files with a path length that exceeds 256 characters, the compiled message flow will not be generated and cannot be deployed. Therefore, make sure that the names of your ESQL files, mapping files, projects, and broker schema are as short as possible.

An ESQL file can also be created automatically for you. If you select Open ESQL from the menu displayed when you right-click a Compute, Database, or Filter node, and the module identified by the appropriate property does not already exist within the broker schema, a module is automatically created for you. This is created in the file <message_flow_name>.esql in the same broker schema within the same project as the <message_flow_name>.msgflow file. If that ESQL file does not already exist, that is also created for you.

The contents of a single ESQL file do not have any specific relationship with message flows and nodes. It is your decision which modules are created in which files (unless the specified module, identified by the appropriate property, is created by default in the file <message_flow_name>.esql as described above). Monitor the size and complexity of the ESQL within each file, and split the file if it becomes difficult to view or manage.

If you create reusable subroutines (at broker schema level) within an ESQL file, you might want to refer to these routines from ESQL modules in another project. To do this, specify that the project that wants to invoke the subroutines depends on the project in which the ESQL file containing them is defined. You can specify this when you create the second project, or you can update project dependencies by selecting the project, clicking **Properties**, and updating the dependencies in the Project Reference page of the properties dialog.

Opening an existing ESQL file

You can add to and modify ESQL code that you have created in an ESQL file in a message flow project.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 156

To open an existing ESQL file:

1. Switch to the Broker Application Development perspective.
2. In the Navigator view, double-click the ESQL file that you want to open. The file is opened in the editor view.
3. Work with the contents of file to make your changes. The file can contain modules relating to specific nodes in a message flow, PATH statements, and declarations at broker schema level such as reusable constants and procedures. Scroll through the file to find the specific content that you want to work with.
4. You can select the content that you want to work with by selecting its name in the Outline view. The code for the selected resource is highlighted.

You can also open an ESQL file when you have a message flow open in the editor view by selecting an appropriate node (of type Compute, Database, or Filter), right-clicking, and selecting **Open ESQL**. In this case, the ESQL file that contains this module is opened, and the module for the selected node is highlighted in the editor view.

Creating ESQL for a node

Create ESQL to customize the behavior of a Compute, Database, or Filter node within an ESQL file.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 156

Within the ESQL file, create a module that is associated with a node in your message flow. A module can be associated with only one node of a particular type (Compute, Database, or Filter). Within the module you can create and use functions and procedures as well as the supplied statements and functions. You can also create local constants and variables.

If you have created constants, functions, or procedures at the broker schema level, you can also refer to these within the module. You can define routines at a level at which many different modules can use them, which can save you development time and maintenance effort.

To create ESQL for a node:

1. Switch to the Broker Application Development perspective.
2. In the Navigator view, double-click the message flow that includes the node for which you want to create ESQL. The message flow opens in the editor view.

- Right-click the node (which must be Compute, Database, or Filter) and click **Open ESQL**. The default ESQL file for this message flow, `<message_flow_name>.esql`, is opened in the editor view (the file is created if it does not already exist).

(If you have already created the default file and you click **Open ESQL**, the file is opened in the editor view and a new module is created and highlighted.) A skeleton module is created for this node at the end of the ESQL file. Its exact content depends on the type of node.

The following module is created for a Compute node:

```
CREATE COMPUTE MODULE <module_name>
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    -- CALL CopyMessageHeaders();
    -- CALL CopyEntireMessage();
    RETURN TRUE;
  END;

  CREATE PROCEDURE CopyMessageHeaders() BEGIN
    DECLARE I INTEGER 1;
    DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
    WHILE I < J DO
      SET OutputRoot.*[I] = InputRoot.*[I];
      SET I = I + 1;
    END WHILE;
  END;

  CREATE PROCEDURE CopyEntireMessage() BEGIN
    SET OutputRoot = InputRoot;
  END;
END MODULE;
```

To make the preceding ESQL deployable to a Version 2.1 broker, you must pass the `InputRoot` and `OutputRoot` module level variables to the procedure or function as shown by the bold text in the following example:

```
CREATE COMPUTE MODULE <module_name>
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    -- CALL CopyMessageHeaders(InputRoot, OutputRoot);
    -- CALL CopyEntireMessage(InputRoot, OutputRoot);
    RETURN TRUE;
  END;

  CREATE PROCEDURE CopyMessageHeaders(IN InputRoot REFERENCE, IN
  OutputRoot REFERENCE) BEGIN
    DECLARE I INTEGER 1;
    DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
    WHILE I < J DO
      CREATE LASTCHILD OF OutputRoot DOMAIN FIELDNAME (
      InputRoot.*[I]; /*create the parser for OutputRoot*/
      SET OutputRoot.*[I] = InputRoot.*[I];
      SET I = I + 1;
    END WHILE;
  END;

  CREATE PROCEDURE CopyEntireMessage(IN InputRoot REFERENCE, IN
  OutputRoot REFERENCE)) BEGIN
    SET OutputRoot = InputRoot;
  END;
END MODULE;
```

The module name is determined by the value that you have set for the corresponding node property. The default is `<message_flow_name>_<node_type>`. The Main function contains calls to two

procedures, described below, that are declared within the Compute node module following the function Main. These calls are commented out. If you want to include the function that they provide, uncomment these lines and place them at the appropriate point in the ESQL that you create for Main.

CopyMessageHeaders

This procedure loops through the headers contained in the input message and copies each one to the output message.

If you are migrating from Version 2.1, this procedure is equivalent to the code generated when you select the Copy message headers button on the Compute node properties dialog.

CopyEntireMessage

This procedure copies the entire contents of the input message, including the headers, to the output message.

If you are migrating from Version 2.1, this procedure is equivalent to the code generated when you select the Copy entire message button on the Compute node properties dialog.

If you create an ESQL module for a Database node, the following module is created:

```
CREATE DATABASE MODULE <module_name>
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    RETURN TRUE;
  END;
END MODULE;
```

For a Filter node, the module is identical to that created for the Database node except for the first line, which reads:

```
CREATE FILTER MODULE <module_name>
```

4. Add ESQL to this file to customize the behavior of the node.

You should start by adding ESQL statements within the Main function, that is after the BEGIN statement, and before RETURN TRUE. You can add DECLARE statements within the module that are not within the Main function. To add a new line into the file, press Enter.

To help you to code valid ESQL, the editor displays a list of valid statements and functions at the point of the cursor. To invoke this assistance, click **Edit** → **Content Assist**. On some systems, you might also be able to use the key combination Ctrl+Space. Scroll through the list displayed to find and highlight the one that you want, and press Enter. The appropriate code is inserted into your module, and the list disappears.

Content assistance is provided in the following areas:

- Applicable keywords, based on language syntax.
- Blocks of code that go together, such as BEGIN END;.
- Constants that you have defined, identifiers, labels, functions, and procedures that can be used, where the routines can be in any projects, even if these are not referenced by the current project.
- Database schema and table names after the database correlation name, as well as table column names in INSERT, UPDATE, DELETE, and SELECT statements, and, in most cases, the WHERE clauses of those statements.

- Elements of message field reference: runtime domain (parser) names, format of type expression, namespace identifiers, namespace-qualified element and attribute names, and format of index expression.
- Content in the Properties folder under the output message root.
- For the DECLARE NAMESPACE statement, target namespaces of message sets and schema names.

Content assistance works only if the ESQL can be parsed correctly. Errors such as END missing after BEGIN, and other unterminated block statements, cause parser failures and no content assistance is provided. Try content assistance in other areas around the statement where it does not work to narrow down the point of error. Alternatively, save the ESQL file; saving the file causes validation and all syntax errors are written to the Tasks view. Refer to the errors reported to understand and correct the ESQL syntax. If you use content assistance to generate most statements (such as block statements), these are correctly entered and there is less opportunity for error.

5. When you have finished working with this module, you can close the ESQL file. Save the file before you close it to retain all your changes and validate your ESQL.

If you prefer, you can open the ESQL file directly and create the module within that file using the editor. To do this:

1. Switch to the Broker Application Development perspective.
2. Select the ESQL file in which you want to create the module. Either double-click to open this file in the editor view, or right-click and click **Open**.
3. In the editor view, position your cursor on a new line and use content assistance to select the appropriate module skeleton for this type of node, for example CREATE COMPUTE MODULE END MODULE;. You can type this in yourself if you prefer, but you must ensure that what you type is consistent with the required skeleton, shown above. You are recommended to use content assistance because this gives you additional help by inserting only valid ESQL, and by inserting matching end statements (for example, END MODULE;) where these are required.
4. Complete the coding of the module as appropriate.

Whichever method you use to open the ESQL file, be aware that the editor provides functions to help you to code ESQL. This section refers to content assistance but there are further functions available in the editor. For information about these functions, see ESQL editor.

Modifying ESQL for a node

If you want to change the customization of a node that requires ESQL (Compute, Database, or Filter), you can modify the ESQL statements within the module that you created for that node.

Before you start

To complete this task, you must have completed the following task:

- “Creating ESQL for a node” on page 158

To modify ESQL code:

1. Switch to the Broker Application Development perspective.
2. In the Navigator view, select the message flow that you want to work with and double-click it. The message flow is opened in the editor view.

3. Right-click the node corresponding to the ESQL module that you want to modify and click **Open ESQL**. The ESQL file is opened in the editor view. The module for this node is highlighted.
4. Make the changes that you want in the module, by entering new statements (remember that you can use Content Assist, available from the Edit menu or, on some systems, by pressing Ctrl+Space), changing existing statements by overtyping, or deleting statements using the Delete or backspace keys. Note that, to get Content Assist to work with message references, you must set up a project reference from the project containing the ESQL to the project containing the message set. For information about setting up a project reference, see Project references.
5. You can change the name of the module that you are working with, by over-typing the current name with the new one. Remember that, if you do that, you must also change the node property *ESQL Module* to reflect the new name to ensure that the correct ESQL code is deployed with the node.
6. When you have finished working with this module, you can close the ESQL file. Save the file before you close it to retain all your changes and validate your ESQL.

If you prefer, you can open the ESQL file directly by double-clicking it in the Navigator view. You can select the module that you want to work with from the Outline view.

The editor provides functions that you can use to help you modify your ESQL code. These functions are described in ESQL editor.

You can also modify the ESQL source by selecting **Source** → **Format**. This option formats all selected lines of code (unless only partially selected, when they are ignored), or, if no lines are selected, formats the entire file (correcting alignments and indentation).

Adding comments to ESQL:

You can add comments to and remove comments from your ESQL code:

1. To change an existing line of code into a comment line, click **Source** → **Comment**.
2. To change a comment line to a code line, click **Source** → **Uncomment**.
3. To create a new comment line, press Enter to create a new line and either type the comment identifier `--` or click **Source** → **Comment**. You can enter any text after the identifier: everything you type is ignored by the ESQL editor.

Saving an ESQL file

When you edit your ESQL file, you can save it both to preserve the additions and modifications that you have made and to force the editor to validate the file's content.

Before you start

To complete this task, you must have completed the following task:

- "Creating an ESQL file" on page 156

To save an ESQL file:

1. Switch to the Broker Application Development perspective.


2. Create a new ESQL file or open an existing ESQL file.
3. Make the changes to the contents of the ESQL file.
4. When you have finished working, save the file to retain all your changes by clicking **File** → **Save <filename>.esql** or **File** → **Save All** (the menu always shows the current filename correctly).

When you save the file, the validator is invoked by the editor to check that the ESQL obeys all grammar and syntax rules (specified by the syntax diagrams and explanations in “ESQL reference” on page 779).


You can request additional validation when you set ESQL preferences. Click **Window** → **Preferences**. The Preferences dialog is displayed:

5. Expand the item for ESQL and Mapping on the left and click Validation. You can choose a value of warning (the default), error, or ignore for the following four categories of error:
 - a. Unresolved identifiers
 - b. Message references do not match message definitions
 - c. Database references do not match database schema
 - d. Use of deprecated keywords

Validating message definitions can impact response times in the editor, particularly if you have complicated ESQL that makes many references to a complex message definition. You might choose to delay this validation. However, you are recommended to invoke it when you have finished developing the message flow and are about to deploy it, to avoid runtime errors. For each error found, the editor writes an entry in the Tasks view, providing both the code line number and the reason for the error.

6. If you double-click the error, the editor positions your cursor on the line in which it found that error. The line is also highlighted by the error icon  in the margin to the left.

The editor might also find potential error situations, that it highlights as

warnings (with the warning icon ), which it also writes to the tasks view. For example, you might have included a `BROKER SCHEMA` statement that references an invalid schema (namespace).

Check your code, and make the corrections required by that statement or function.

Save As:

You can save a copy of this ESQL file by using **File** → **Save As...**

1. Click **File** → **Save <name> As...**
2. Specify the message flow project in which you want to save a copy of the ESQL file. The project name defaults to the current project. You can accept this name, or choose another name from the valid options that are displayed in the File Save dialog.
3. Specify the name for the new copy of the ESQL file. If you want to save this ESQL file in the same project, you must either give it another name, or confirm that you want to overwrite the current copy (that is, copy the file to itself).

If you want to save this ESQL file in another project, the project must already exist (you can only select from the list of existing projects). You can save the file with the same or another name in another project.

4. Click **OK**. The message flow is saved and the message flow editor validates its contents. The editor provides a report of any errors that it finds in the Tasks view.

Copying an ESQL file

You might find it useful to copy an ESQL file as a starting point for a new ESQL file that has similar function.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 156

To copy an ESQL file:

1. Switch to the Broker Application Development perspective.
2. In the Navigator view, select the ESQL file (<message_flow_name>.esql) that you want to copy. Right-click the file and click **Copy** from the menu.
3. Right-click the broker schema within the message flow project to which you want to copy the ESQL file and click **Paste**. You can copy the ESQL file to the same broker schema within the same message flow project, or to a different broker schema within the same message flow project, or to a broker schema in a different message flow project.

When you copy an ESQL file, the associated files (message flow, and mapping if present) are not automatically copied to the same target message flow project. If you want these files copied as well, you must do this explicitly following this procedure.

If you want to use this ESQL file with another message flow, ensure that the modules within the ESQL file match the nodes that you have in the message flow, and that the node properties are set correctly.

You can also use **File** → **Save As** to copy an ESQL file. This is described in “Saving an ESQL file” on page 162.

Renaming an ESQL file

You can rename an ESQL file within the message flow project. You might want to do this, for example, if you have renamed the message flow with which it is associated.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 156

To rename an ESQL file:

1. Switch to the Broker Application Development perspective.
2. In the Navigator view, right-click the ESQL file that you want to rename. Its default name is <message_flow_name>.esql. Click **Rename** or click **File** → **Rename**. If you have selected the ESQL file, you can press F2. The Rename Resource dialog is displayed.
3. Enter the new name for the ESQL file. Click **OK** to complete the action, or **Cancel** to cancel the request. If you click **OK**, the ESQL file is renamed.

When the rename is done, any references that you have to this ESQL file are no longer valid and you must correct them. If you are unsure where the references are, click **File** → **Save All**. This saves and validates all resources. Unresolved references are listed in the Tasks view, and you can click each error listed to locate and update the references.

Moving an ESQL file

If you move a message flow from one broker schema to another, or from one project to another, you might want to move any ESQL file that is associated with that message flow.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 156



To move an ESQL file:

1. Switch to the Broker Application Development perspective.
2. Move the ESQL file in one of the following ways:
 - a. Drag and drop the ESQL file that you want to move from its current location to a broker schema within the same or another message flow project.

If the target location that you have chosen is not valid (for example, if an ESQL file of this name already exists in the broker schema), the invalid icon is displayed and the move is not completed.
 - b. Right-click the ESQL file and click **Move**, or click **File** → **Move**. The Move dialog is displayed.

Select the project and the broker schema from the list of valid targets that is shown in the dialog.

Click **OK** to complete the move, or **Cancel** to cancel the request.

If you click **OK**, the ESQL file is moved to its new location.
3. Check the Tasks view for any errors (indicated by the error icon ) or warnings (indicated by the warning icon ) generated by the move.

The errors in the Tasks view include those caused by broken references. When the move is completed, all references to this ESQL file are checked. If you have moved the file within the same named broker schema within the same message flow project, all references are still valid. If you have moved the file to another broker schema in the same or another message flow project, the references are broken. If you have moved the file to the same named broker schema in another message flow project, the references might be broken if the project references are not set correctly to recognize external references in this file. These errors occur because resources are linked by a fully-qualified name.
4. Double-click each error or warning to correct it. This opens the message flow that has the error in the editor view and highlights the node in error.

When you move an ESQL file, its associated files (for example, the message flow file) are not automatically moved to the same target broker schema. You must move these files yourself.

Changing ESQL preferences

You can modify the way in which ESQL is generated, displayed in the editor, and validated by the editor:

- “Setting the ESQL code generation level”
- “Changing ESQL editor settings”
- “Changing ESQL validation settings” on page 167

Setting the ESQL code generation level:

You can specify the level of ESQL runtime code that is generated when you add a message flow to a bar file:

- If your ESQL code includes references to databases, you can specify what schema name is used to identify the database tables when the runtime code is generated.
- The code that is generated must be compatible with the broker to which the bar file is deployed. If you have brokers at Version 2.1 and Version 5.0 levels in your broker domain, you must ensure that the message flows that you deploy to a broker can be executed by that broker.

To change the ESQL code generation level:

1. Switch to the Broker Application Development perspective.
2. Click **Window** → **Preferences**. The Preferences dialog is displayed.
3. Expand the item for ESQL and Mapping on the left and click Code Generation.
4. Update the settings for code generation. See ESQL editor for details of the settings and their values.
5. When you have completed your changes, click **Apply** to close the Preferences dialog, apply your changes and leave the Preferences dialog open. Click **OK** to apply your changes and close the dialog. Click **Cancel** to close the dialog and discard your changes.
6. If you want to return your ESQL editor preferences to the initial values, click **Restore Defaults**. All values are reset to the original settings.

The changes are implemented when you add a message flow to a bar file.

Changing ESQL editor settings:

When you open an ESQL file in the editor view, you can tailor the editor appearance by changing editor settings.

To change ESQL editor settings:

1. Switch to the Broker Application Development perspective.
2. Click **Window** → **Preferences** → **Workbench** → **Colors and Fonts** → **ESQL Editor Text Font**. The Preferences dialog is displayed.
3. Update the settings available for fonts and colors:
 - Click the General tab to change text font and the displayed tab width within the ESQL editor.
 - Click the Colors tab to change the color of the editor view background, and of the entities displayed in the editor view. These include comments and keywords within your ESQL code.

4. When you have completed your changes, click **Apply** to close the Preferences dialog, apply your changes and leave the Preferences dialog open. Click **OK** to apply your changes and close the dialog. Click **Cancel** to close the dialog and discard your changes.
5. If you want to return your ESQL editor settings to the initial values, click **Restore Defaults**. All values are reset to the original settings.

If you change the editor settings when you have an editor session active, the changes are implemented immediately. If you do not have an editor session open, you see the changes when you next edit an ESQL file.

Changing ESQL validation settings:

You can specify the level of validation that the ESQL editor performs when you save a .esql file. If the validation you have requested results in warnings, you can deploy a bar file containing this message flow. However, if errors are reported, you cannot deploy the bar file.

To change ESQL validation settings:

1. Switch to the Broker Application Development perspective.
2. Click **Window** → **Preferences**. The Preferences dialog is displayed.
3. Expand the item for ESQL and Mapping on the left and click Validation.
4. Update the settings for what is validated, and for what warnings or errors are reported. See ESQL editor for details of the settings and their values.
5. When you have completed your changes, click **Apply** to close the Preferences dialog, apply your changes and leave the Preferences dialog open. Click **OK** to apply your changes and close the dialog. Click **Cancel** to close the dialog and discard your changes.
6. If you want to return your ESQL editor preferences to the initial values, click **Restore Defaults**. All values are reset to the original settings.

If you make changes to the validation settings, the changes are implemented immediately for currently open edit sessions and for subsequent edit sessions.

Deleting ESQL for a node

If you delete a node from a message flow, you can delete the ESQL module that you created to customize its function.

Before you start

To complete this task, you must have completed the following task:

- “Creating ESQL for a node” on page 158

To delete ESQL code:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with by double-clicking it in the Navigator view. The message flow is opened in the editor view.
3. Select the node for which you want to delete the ESQL module, right-click and click **Open ESQL**. The ESQL file is opened in the editor view, with the module for this node highlighted.
4. Press the Delete or backspace key to delete the whole module.

5. When you have finished working with this module, you can close the ESQL file. Save the file before you close it to retain all your changes. Save also validates your ESQL: see “Saving an ESQL file” on page 162.

If you prefer, you can open the ESQL file directly by double-clicking it in the Navigator view. The ESQL file is opened in the editor view. Select the module that you want to delete from the Outline view and delete it as described above. You can also right-click on the module name in the Navigator view (the modules in the ESQL file are visible if you expand the view of the file by clicking the + beside the file name) and click **Delete**.

Deleting an ESQL file

If you delete a message flow, or if you have deleted all the ESQL code in an ESQL file, you can delete the ESQL file.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 156

To delete an ESQL file:

1. Switch to the Broker Application Development perspective.
2. Within the Navigator view, right-click the ESQL file that you want to delete, and click **Delete**. A dialog is displayed that asks you to confirm the deletion.

You can also select the file in the Navigator view, and click **Edit** → **Delete**. A dialog is displayed that asks you to confirm the deletion.

3. Click **Yes** to delete the file, or **No** to cancel the delete request.

If you maintain resources in a shared repository, a copy is retained in that repository. You can follow the instructions provided by the repository supplier to retrieve the file if required.

If you are using the local file system or a shared file system to store your resources, no copy of the file is retained. Be careful to select the correct file when you complete this task.

Writing ESQL

When you create a message flow, you include input nodes that receive the messages and, optionally, output nodes that send out new or updated messages. If required by the processing that must be performed on the message, you can include other nodes after the input node that complete the actions that your applications need.

Some of the built-in nodes allow you to customize the processing that they provide. The Compute, Database, and Filter nodes require you to provide a minimum level of ESQL, and you can provide much more than the minimum to control precisely the behavior of each node. This set of topics discusses ESQL and the ways in which you can use it to customize these nodes.

The DataDelete, DataInsert, DataUpdate, Extract, Mapping, and Warehouse nodes provide a mapping interface with which you can customize their function. The ways in which you can use the mapping functions associated with these nodes are described in “Developing message mappings” on page 302.

ESQL provides a rich and flexible syntax for statements and functions that let you check and manipulate message and database content. You can:

- Read the contents of the input message
- Modify message content with data from databases
- Modify database content with data from messages
- Construct new output messages created from all, part, or none of the input message (in the Compute node only)

The following topics provide more information about these and other tasks that you can perform with ESQL. Unless otherwise stated, these guidelines apply to messages in all message domains except the BLOB domain, for which you can implement a limited set of actions.

- “Tailoring ESQL code for different node types” on page 170
- “Manipulating message body content” on page 171
- “Manipulating other parts of the message tree” on page 189
- “Transforming from one data type to another” on page 197
- “Adding keywords to ESQL files” on page 204
- “Accessing databases from ESQL” on page 204
- “Coding ESQL to handle errors” on page 214
- “Accessing broker properties from ESQL” on page 284
- “Configuring a message flow at deployment time using UDPs” on page 284

The following topics provide additional information specific to the parser that you have specified for the input message:

- “Manipulating messages in the MRM domain” on page 220
- “Manipulating messages in the XML domain” on page 239
- “Manipulating messages in the XMLNS domain” on page 267
- “Manipulating messages using the XMLNSC parser” on page 269
- “Manipulating messages in the JMS domains” on page 275
- “Manipulating messages in the IDoc domain” on page 275
- “Manipulating messages in the MIME domain” on page 275
- “Manipulating messages in the BLOB domain” on page 277

ESQL examples

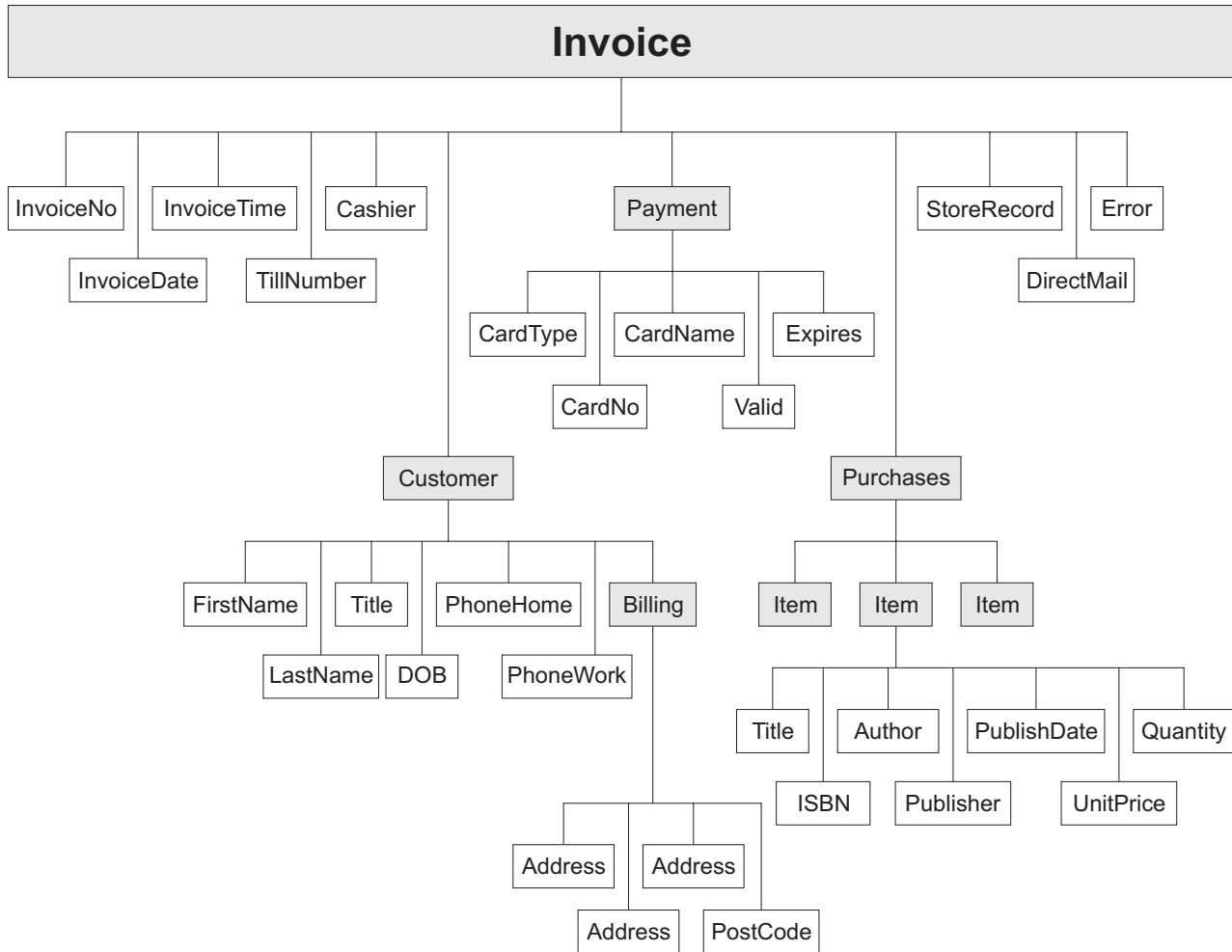
Most of the examples included in the topics listed above show parser-independent ESQL. If examples include reference to MRM, they assume that you have modeled the message in the MRM and that you have set the names of the MRM objects to be identical to the names of the corresponding tags or attributes in the XML source message. Some examples are also shown for the XML domain. Unless stated otherwise, the principals illustrated are the same for all message domains. For domain-specific information, refer to the appropriate link in the list above.

Most of the topics that include example ESQL use the ESQL sample message, *Invoice*, as the input message to the logic. This message is provided in XML source format (with tags and attributes) in “Example message” on page 991, and is shown below in diagrammatic form.

The topics specific to the MRM domain use the message that is created in the Video Rental sample sample.

A few other input messages are used to show ESQL that provides function on messages with a structure or content that is not included in the Invoice or Video

samples. Where this occurs, the input message is included in the topic that refers to it.



Tailoring ESQL code for different node types

When you code ESQL to configure Compute, Database, and Filter node behavior, be aware of the limitations of each type of node:

Compute node

You can configure the Compute node to do any of the following operations:

- Update data in a database.
- Insert data into a database.
- Delete data from a database.
- Update the Environment tree.
- Update the LocalEnvironment tree.
- Create one or more output messages, with none, some, or all the content of the input message, and propagate these new messages to the next node in the message flow.

If you want to propagate the input LocalEnvironment to the output LocalEnvironment, remember to set the Compute node property *Compute mode* to an appropriate value. The Environment is always propagated in the output message.

Database node

You can configure the Database node to do any of the following operations:

- Update data in a database.
- Insert data into a database.
- Delete data from a database.
- Update the Environment tree.
- Update the LocalEnvironment tree.
- Propagate the input message to the next node in the message flow.

Filter node

You can configure the Filter node to do any of the following operations:

- Update data in a database.
- Insert data into a database.
- Delete data from a database.
- Update the Environment tree.
- Update the LocalEnvironment tree.
- Propagate the input message to the next node in the message flow (the terminal through which the message is propagated depends on the result of the filter expression).

View the remaining tasks in this section to find the details of how you can perform these operations.

Manipulating message body content

The message body is always the last child of root, and is identified by its parser name, for example XML or MRM.

The following topics describe how you can refer to, modify, and create message body data. The information provided here is domain independent.

- “Referencing field types”
- “Accessing elements in the message body” on page 172
- “Accessing known multiple occurrences of an element” on page 176
- “Accessing unknown multiple occurrences of an element” on page 177
- “Using anonymous field references” on page 178
- “Creating dynamic field references” on page 179
- “Creating new fields” on page 180
- “Generating multiple output messages” on page 182
- “Using numeric operators with datetime values” on page 183
- “Calculating a time interval” on page 184
- “Selecting a subfield from a larger field” on page 185
- “Copying repeating fields” on page 186
- “Manipulating repeating fields in a message tree” on page 188

Referencing field types:

Some message parsers have complex models in which it is not enough to identify a field simply by its name and an array subscript. In these cases, you associate an optional field type with an element of data in the tree format.

Each element within the parsed tree can be one of three types:

Name element

A name element has a string, which is the name of the element, associated with it. An example of a name element is `XMLElement`, described in “XML element” on page 767.

Value element

A value element has a value associated with it. An example of a value element is `XMLContent`, described in “XML content” on page 767.

Name-value element

A name-value element is an optimization of the case where a name element contains only a value element and nothing else. The element contains both a name and a value. An example of a name-value element is `XMLAttribute`, described in “XML attribute” on page 766.

Accessing elements in the message body:

When you want to access the contents of a message, for reading or writing, use the structure and arrangement of the elements in the tree that is created from the input bit stream by the parser. Follow the relevant parent and child relationships from the top of the tree downwards, until you reach the required element.

- If you are referring to the input message tree to interrogate its content in a Compute node, use correlation name `InputBody` followed by the path to the element to which you are referring. `InputBody` is equivalent to `InputRoot` followed by the parser name (for example, `InputRoot.MRM`), which you can use if you prefer.
- If you are referring to the output message tree to set or modify its content in the Compute node, use correlation name `OutputRoot` followed by the parser name (for example, `OutputRoot.MRM`).
- If you are referring to the input message to interrogate its contents in a Database or Filter node, use correlation name `Body` to refer to the start of the message. `Body` is equivalent to `Root` followed by the parser name (for example, `Root.XML`), which you can use if you prefer.

You must use these different correlation names because there is only one message to which to refer in a Database or Filter node; you cannot create a new output message in these nodes. Use a Compute node to create a new output message.

When you construct field references, the names that you use must be valid ESQL identifiers that conform to ESQL rules. If you enclose anything in double quotation marks, ESQL interprets it as an identifier. If you enclose anything in single quotation marks, ESQL interprets it as a character literal. You must enclose all strings (character strings, byte strings, or binary (bit) strings) in quotation marks, as shown in the examples below. To include a single or double quotation mark within a string, include two consecutive single or double quotation marks.

Important: For a full description of field reference syntax, see “ESQL field references” on page 792.

For more information about ESQL data types, see “ESQL data types in message flows” on page 780.

Assume that you have created a message flow that handles the message Invoice, shown in the figure in “Writing ESQL” on page 168. If, for example, you want to interrogate the element CardType from within a Compute node, use the following statement:

```
IF InputBody.Invoice.Payment.CardType='Visa' THEN
  DO;
  -- more ESQL --
END IF;
```

If you want to make the same test in a Database or Filter node (where the reference is to the single input message), code:

```
IF Body.Invoice.Payment.CardType='Visa' THEN
  DO;
  -- more ESQL --
END IF;
```

If you want to copy an element from an input XML message to an output message in the Compute node without changing it, use the following ESQL:

```
SET OutputRoot.XML.Invoice.Customer.FirstName =
  InputBody.Invoice.Customer.FirstName;
```

If you want to copy an element from an input XML message to an output message and update it, for example by folding to uppercase or by calculating a new value, code:

```
SET OutputRoot.XML.Invoice.Customer.FirstName =
  UPPER(InputBody.Invoice.Customer.FirstName);
SET OutputRoot.XML.Invoice.InvoiceNo = InputBody.Invoice.InvoiceNo + 1000;
```

If you want to set a STRING element to a constant value, code:

```
SET OutputRoot.XML.Invoice.Customer.Title = 'Mr';
```

You can also use the equivalent statement:

```
SET OutputRoot.XML.Invoice.Customer.Title VALUE = 'Mr';
```

If you want to update an INTEGER or DECIMAL, for example the element TillNumber, with the value 26, use the following assignment (valid in the Compute node only):

```
SET OutputRoot.MRM.Invoice.TillNumber=26;
```

The integer data type stores numbers using the 64-bit twos complement form, allowing numbers that range from -9223372036854775808 to 9223372036854775807. You can specify hexadecimal notation for integers as well as normal integer literal format. The hexadecimal letters A to F can be written in upper or lower case, as can the X after the initial zero, which is required. The example below produces the same result as the example shown above:

```
SET OutputRoot.MRM.Invoice.TillNumber= 0x1A;
```

The following examples show SET statements for element types that do not appear in the example Invoice message.

To set a FLOAT element to a non-integer value, code:

```
SET OutputRoot.MRM.FloatElement1 = 1.2345e2;
```

To set a BINARY element to a constant value, code:

```
SET OutputRoot.MRM.BinaryElement1 = X'F1F1';
```

For BINARY values, you must use an initial character X (upper or lower case) and enclose the hexadecimal characters (also upper or lower case) in single quotation marks, as shown.

To set a BOOLEAN element to a constant value (the value 1 equates to true, 0 equates to false), code:

```
SET OutputRoot.MRM.BooleanElement1 = true;
```

or

```
SET OutputRoot.MRM.BooleanElement1 = 1;
```

You can use the SELECT statement to filter records from an input message without reformatting the records, and without any knowledge of the complete format of each record. Consider the following example:

```
-- Declare local variable
DECLARE CurrentCustomer CHAR 'Smith';

-- Loop through the input message
SET OutputRoot.XML.Invoice[] =
    (SELECT I FROM InputRoot.XML.Invoice[] AS I
     WHERE I.Customer.LastName = CurrentCustomer
    );
```

This writes all records from the input Invoice message to the output message if the WHERE condition (LastName = Smith) is met. All records that do not meet the condition are not copied from input to output. I is used as an alias for the correlation name InputRoot.XML.Invoice[].

The declared variable CurrentCustomer is initialized on the DECLARE statement: this is the most efficient way of declaring a variable for which the initial value is known.

You can use this alias technique with other SELECT constructs. For example, if you want to select all the records of the input Invoice message, and create an additional record:

```
-- Loop through the input message
SET OutputRoot.XML.Invoice[] =
    (SELECT I, 'Customer' || I.Customer.LastName AS ExtraField
     FROM InputRoot.XML.Invoice[] AS I
    );
```

You could also include an AS clause to place records in a subfolder in the message tree:

```
-- Loop through the input message
SET OutputRoot.XML.Invoice[] =
    (SELECT I AS Order
     FROM InputRoot.XML.Invoice[] AS I
    );
```

If you are querying or setting elements that contain, or might contain, null values, be aware of the following considerations:

Querying null values

When you compare an element to the ESQL keyword `NULL`, this tests whether the element is present in the logical tree that has been created from the input message by the parser.

For example, you can check if an invoice number is included in the current Invoice message with the following statement:

```
IF InputRoot.XML.Invoice.InvoiceNo IS NULL THEN
  DO;
  -- more ESQL --
END IF;
```

You can also use an ESQL reference. The following example illustrates this.

```
DECLARE cursor REFERENCE TO InputRoot.MRM.InvoiceNo;

IF LASTMOVE(cursor) = FALSE THEN
  SET OutputRoot.MRM.Analysis = 'InvoiceNo does not exist in logical tree';
ELSEIF FIELDVALUE(cursor) IS NULL THEN
  SET OutputRoot.MRM.Analysis =
    'InvoiceNo does exist in logical tree but is defined as an MRM NULL value';
ELSE
  SET OutputRoot.MRM.Analysis = 'InvoiceNo does exist and has a value';
END IF;
```

For more information about declaring and using references, see “Creating dynamic field references” on page 179. For a description of the `LASTMOVE` and `FIELDVALUE` functions, see “`LASTMOVE` function” on page 935 and “`FIELDTYPE` function” on page 930.

If the message is in the MRM domain, there are additional considerations for querying null elements that depend on the physical format. For further details, see “Querying null values in a message in the MRM domain” on page 229.

Setting null values

There are two statements that you can use to set null values.

1. If you set the element to `NULL` using the following statement, the element is deleted from the message tree:

```
SET OutputRoot.XML.Invoice.Customer.Title = NULL;
```

If the message is in the MRM domain, there are additional considerations for null values that depend on the physical format. For further details, see “Setting null values in a message in the MRM domain” on page 229.

This is called implicit null processing.

2. If you set the value of this element to `NULL` as follows:

```
SET OutputRoot.XML.Invoice.Customer.Title VALUE = NULL;
```

the element is not deleted from the message tree. Instead, a special value of `NULL` is assigned to the element.

```
SET OutputRoot.XML.Invoice.Customer.Title = NULL;
```

If the message is in the MRM domain, the content of the output bit stream depends on the settings of the physical format null handling properties. For further details, see “Setting null values in a message in the MRM domain” on page 229.

This is called explicit null processing.

If you set an MRM complex element or an XML, XMLNS, or JMS parent element to NULL without using the VALUE keyword, that element and all its children are deleted from the logical tree.

Accessing known multiple occurrences of an element:

When you refer to or create the content of messages, it is very likely that the data contains repeating fields. If you know how many instances there are of a repeating field, and you want to access a specific instance of such a field, you can use an array index as part of a field reference.

For example, you might want to filter on the first line of an address, to expedite the delivery of an order. Three instances of the element Billing.Address are always present in the sample message. To test the first line, write an expression such as:

```
IF Body.Invoice.Customer.Billing.Address[1] = 'Patent Office' THEN
  DO;
  -- more ESQL --
END IF;
```

The array index [1] indicates that it is the first instance of the repeating field that you are interested in (array indices start at 1). An array index such as this can be used at any point in a field reference, so you could, for example, filter on the following test:

```
IF Body.Invoice."Item"[1].Quantity > 2 THEN
  DO;
  -- more ESQL --
END IF;
```

You can refer to the last instance of a repeating field using the special [<] array index, and to instances relative to the last (for example, the second to last) as follows:

- Field[<] indicates the last element.
- Field[<1] indicates the last element.
- Field[<2] indicates the last but one element (the penultimate element).

You can also use the array index [>] to represent the first element, and elements relative to the first element in a similar way.

- Field[>] indicates the first element. This is equivalent to Field[1].

The following examples refer to the Invoice message using these indexes:

```
IF Body.Invoice.Customer.Billing.Address[<] = 'Hampshire' THEN
  DO;
  -- more ESQL --
END IF;
IF Body.Invoice.Customer.Billing.Address[<2 ] = 'Southampton' THEN
  DO;
  -- more ESQL --
END IF;
```


You can also use these special indexes for elements that repeat an unknown number of times.

Deleting repeating fields:

If you pass a message with several repeats of an element through a message flow and you want to delete some of the repeats, be aware that the numbering of the repeats is reordered after each delete. For example, if you have a message with five repeats of a particular element, and in the message flow you have the following ESQL:

```
SET OutputRoot.MRM.e_PersonName[1] = NULL;  
SET OutputRoot.MRM.e_PersonName[4] = NULL;
```

You might expect elements one and four to be deleted. However, because repeating elements are stored on a stack, when you delete one, the one above it takes its place. This means that, in the above example, elements one and five are deleted. To avoid this problem, delete in reverse order, that is, delete element four first, then delete element one.

Accessing unknown multiple occurrences of an element:

You are very likely to deal with messages that contain repeating fields with an unknown number of repeats. This is the situation with the `Item` field in the example message in “Example message” on page 991.

To write a filter that takes into account all instances of the `Item` field, you need to use a construct that can iterate over all instances of a repeating field. The quantified predicate allows you to execute a predicate against all instances of a repeating field, and collate the results.

For example, you might want to verify that none of the items that are being ordered has a quantity greater than 50. To do this you could write:

```
FOR ALL Body.Invoice.Purchases."Item" []  
  AS I (I.Quantity <= 50)
```

With the quantified predicate, the first thing to note is the brackets `[]` on the end of the field reference after `FOR ALL`. These tell you that you are iterating over all instances of the `Item` field.

In some cases, this syntax appears unnecessary because you can get that information from the context, but it is done for consistency with other pieces of syntax.

The `AS` clause associates the name `I` with the current instance of the repeating field. This is similar to the concept of iterator classes used in some object oriented languages such as C++. The expression in parentheses is a predicate that is evaluated for each instance of the `Item` field.

A description of this example is:

Iterate over all instances of the field `Item` inside `Body.Invoice`. For each iteration:

1. Bind the name `I` to the current instance of `Item`.
2. Evaluate the predicate `I.Quantity <= 50`. If the predicate:
 - Evaluates to `TRUE` for all instances of `Item`, return `TRUE`.

- Is FALSE for any instance of Item, return FALSE.
- For a mixture of TRUE and UNKNOWN, return UNKNOWN.

The above is a description of how the predicate is evaluated if you use the ALL keyword. An alternative is to specify SOME, or ANY, which are equivalent. In this case the quantified predicate returns TRUE if the sub-predicate returns TRUE for any instance of the repeating field. Only if the sub-predicate returns FALSE for all instances of the repeating field does the quantified predicate return FALSE. If a mixture of FALSE and UNKNOWN values are returned from the sub-predicate, an overall value of UNKNOWN is returned.

In the following filter expression:

```
FOR ANY Body.Invoice.Purchases."Item" []
  AS I (I.Title = 'The XML Companion')
```

the sub-predicate evaluates to TRUE. However this next expression returns FALSE:

```
FOR ANY Body.Invoice.Purchases."Item" []
  AS I (I.Title = 'C Primer')
```

because the C Primer is not included on this invoice. If some of the items in the invoice do not include a book title field, the sub-predicate returns UNKNOWN, and the quantified predicate returns the value UNKNOWN.

To deal with the possibility of null values appearing, write this filter with an explicit check on the existence of the field, as follows:

```
FOR ANY Body.Invoice.Purchases."Item" []
  AS I (I.Book IS NOT NULL AND I.Book.Title = 'C Primer')
```

The predicate IS NOT NULL ensures that, if an Item field does not contain a Book, a FALSE value is returned from the sub-predicate.

You can also manipulate arbitrary repeats of fields within a message by using a SELECT expression, as described in “Referencing columns in a database” on page 205.

You can refer to the first and last instances of a repeating field using the [>] and [<] array indexes, and to instances relative to the first and last, even if you do not know how many instances there are. These indexes are described in “Accessing known multiple occurrences of an element” on page 176.

Alternatively, you can use the CARDINALITY function to determine how many instances of a repeating field there are. For example:

```
DECLARE I INTEGER CARDINALITY(Body.Invoice.Purchases."Item" [])
```

Using anonymous field references:

You can refer to the array of all children of a particular element by using a path element of *. So, for example:

```
InputRoot.*[]
```

is a path that identifies the array of all children of InputRoot. This is often used in conjunction with an array subscript to refer to a particular child of an entity by position, rather than by name. For example:

InputRoot.*[<]

Refers to the last child of the root of the input message, that is, the body of the message.

InputRoot.*[1]

Refers to the first child of the root of the input message, the message properties.

You might want to find out the name of an element that has been identified with a path of this kind. To do this, use the FIELDNAME function, which is described in “FIELDNAME function” on page 929.

Creating dynamic field references:

You can use a variable of type REFERENCE as a dynamic reference to navigate a message tree. This acts in a similar way to a message cursor or a variable pointer. It is generally simpler and more efficient to use reference variables in preference to array indexes when you access repeating structures. Reference variables are accepted everywhere. Field references are accepted and come with a set of statements and functions to allow detailed manipulation of message trees.

You must declare a dynamic reference before you can use it. A dynamic reference is declared and initialized in a single statement. The following example shows how to create and use a reference.

```
-- Declare the dynamic reference
DECLARE myref REFERENCE TO OutputRoot.XML.Invoice.Purchases.Item[1];

-- Continue processing for each item in the array
WHILE LASTMOVE(myref)=TRUE
DO
-- Add 1 to each item in the array
  SET myref = myref + 1;
-- Move the dynamic reference to the next item in the array
  MOVE myref NEXTSIBLING;
END WHILE;
```

This example declares a dynamic reference, myref, which points to the first item in the array within Purchases. The value in the first item is incremented by one, and the pointer (dynamic reference) is moved to the next item. Once again the item value is incremented by one. This process continues until the pointer moves outside the scope of the message array (all the items in this array have been processed) and the LASTMOVE function returns FALSE.

The examples below show further examples.

```
DECLARE ref1 REFERENCE TO InputBody.Invoice.Purchases.Item[1];

DECLARE ref2 REFERENCE TO
  InputBody.Invoice.Purchases.NonExistentField;

DECLARE scalar1 CHARACTER;
DECLARE ref3 REFERENCE TO scalar1;
```

In the second example, ref2 is set to point to InputBody because the specified field does not exist.

With the exception of the MOVE statement, which changes the position of the dynamic reference, you can use a dynamic reference anywhere that you can use a static reference. The value of the dynamic reference in any expression or statement is the value of the field or variable to which it currently points. For example, using the message in “Example message” on page 991, the value of Invoice.Customer.FirstName is Andrew. If the dynamic reference myref is set to point at the FirstName field as follows:

```
DECLARE myref REFERENCE TO Invoice.Customer;
```

the value of myref is Andrew. You can extend this dynamic reference as follows:

```
SET myref.Billing.Address[1] = 'Oaklands';
```

This changes the address in the example to Oaklands Hursley Village Hampshire SO213JR.

The position of a dynamic reference remains fixed even if a tree is modified. To illustrate this point the steps that follow use the message in “Example message” on page 991 as their input message and create a modified version of this message as an output message:

1. Copy the input message to the output message.
2. To modify the output message, first declare a dynamic reference ref1 that points at the first item, The XML Companion.

```
DECLARE ref1 REFERENCE TO  
OutputRoot.XML.Invoice.Purchases.Item[1];
```

The dynamic reference is now equivalent to the static reference OutputRoot.XML.Invoice.Purchases.Item[1].

3. Use a create statement to insert a new first item for this purchase.

```
CREATE PREVIOUSIBLING OF ref1 VALUES 'Item';
```

The dynamic reference is now equivalent to the static reference OutputRoot.XML.Invoice.Purchases.Item[2].

Creating new fields:

This topic provides example ESQL code for a Compute node that creates a new output message based on the input message, to which are added a number of additional fields.

The input message received by the Compute node within the message flow is an XML message, and has the following content:

```
<TestCase description="This is my TestCase">  
  <Identifier>ES03B305_T1</Identifier>  
  <Sport>Football</Sport>  
  <Date>01/02/2000</Date>  
  <Type>LEAGUE</Type>  
</TestCase>
```

The Compute node is configured and an ESQL module is created that includes the following ESQL. The code shown below copies the headers from the input message

to the new output message, then creates the entire content of the output message body.

```
-- copy headers
DECLARE i INTEGER 1;
DECLARE numHeaders INTEGER CARDINALITY(InputRoot.*[]);

WHILE i < numHeaders DO
    SET OutputRoot.*[i] = InputRoot.*[i];
    SET i = i + 1;
END WHILE;

CREATE FIELD OutputRoot.XML.TestCase.description TYPE NameValue VALUE 'This is my TestCase';
CREATE FIRSTCHILD OF OutputRoot.XML.TestCase Domain('XML') NAME 'Identifier'
    VALUE InputRoot.XML.TestCase.Identifier;
CREATE LASTCHILD OF OutputRoot.XML.TestCase Domain('XML') NAME 'Sport'
    VALUE InputRoot.XML.TestCase.Sport;
CREATE LASTCHILD OF OutputRoot.XML.TestCase Domain('XML') NAME 'Date'
    VALUE InputRoot.XML.TestCase.Date;
CREATE LASTCHILD OF OutputRoot.XML.TestCase Domain('XML') NAME 'Type'
    VALUE InputRoot.XML.TestCase.Type;
CREATE FIELD OutputRoot.XML.TestCase.Division[1].Number TYPE NameValue
    VALUE 'Premiership';
CREATE FIELD OutputRoot.XML.TestCase.Division[1].Result[1].Number TYPE NameValue VALUE '1';
CREATE FIELD OutputRoot.XML.TestCase.Division[1].Result[1].Home TYPE Name;
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[1].Result[1].Home NAME 'Team'
    VALUE 'Liverpool' ;
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[1].Result[1].Home NAME 'Score'
    VALUE '4';
CREATE FIELD OutputRoot.XML.TestCase.Division[1].Result[1].Away TYPE Name;
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[1].Result[1].Away NAME 'Team'
    VALUE 'Everton';
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[1].Result[1].Away NAME 'Score'
    VALUE '0';

CREATE FIELD OutputRoot.XML.TestCase.Division[1].Result[2].Number TYPE NameValue VALUE '2';
CREATE FIELD OutputRoot.XML.TestCase.Division[1].Result[2].Home TYPE Name;
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[1].Result[2].Home NAME 'Team'
    VALUE 'Manchester United';
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[1].Result[2].Home NAME 'Score'
    VALUE '2';
CREATE FIELD OutputRoot.XML.TestCase.Division[1].Result[2].Away TYPE Name;
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[1].Result[2].Away NAME 'Team'
    VALUE 'Arsenal';
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[1].Result[2].Away NAME 'Score'
    VALUE '3';

CREATE FIELD OutputRoot.XML.TestCase.Division[2].Number TYPE NameValue
    VALUE '2';
CREATE FIELD OutputRoot.XML.TestCase.Division[2].Result[1].Number TYPE NameValue
    VALUE '1';
CREATE FIELD OutputRoot.XML.TestCase.Division[2].Result[1].Home TYPE Name;
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[2].Result[1].Home NAME 'Team'
    VALUE 'Port Vale';
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[2].Result[1].Home NAME 'Score'
    VALUE '9' ;
CREATE FIELD OutputRoot.XML.TestCase.Division[2].Result[1].Away TYPE Name;
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[2].Result[1].Away NAME 'Team'
    VALUE 'Brentford';
CREATE LASTCHILD OF OutputRoot.XML.TestCase.Division[2].Result[1].Away NAME 'Score'
    VALUE '5';
```

The output message that results from the ESQL shown above has the following structure and content:

```

<TestCase description="This is my TestCase">
  <Identifier>ES03B305_T1</Identifier>
  <Sport>Football</Sport>
  <Date>01/02/2000</Date>
  <Type>LEAGUE</Type>
  <Division Number="Premiership">
    <Result Number="1">
      <Home>
        <Team>Liverpool</Team>
        <Score>4</Score>
      </Home>
      <Away>
        <Team>Everton</Team>
        <Score>0</Score>
      </Away>
    </Result>
    <Result Number="2">
      <Home>
        <Team>Manchester United</Team>
        <Score>2</Score>
      </Home>
      <Away>
        <Team>Arsenal</Team>
        <Score>3</Score>
      </Away>
    </Result>
  </Division>
  <Division Number="2">
    <Result Number="1">
      <Home>
        <Team>Port Vale</Team>
        <Score>9</Score>
      </Home>
      <Away>
        <Team>Brentford</Team>
        <Score>5</Score>
      </Away>
    </Result>
  </Division>
</TestCase>

```

Generating multiple output messages:

You can use the PROPAGATE statement to generate multiple output messages in the Compute node. The output messages that you generate can have the same or different content. You can also direct output messages to any of the four alternate output terminals of the Compute node, or to a Label node.

For example, if you want to create three copies of the input message received by the Compute node, and send one to the standard "Out" terminal of the Compute node, one to the first alternate "Out1" terminal of the Compute node, and one to the Label node "ThirdCopy", code the following ESQL:

```

SET OutputRoot = InputRoot;
PROPAGATE;
SET OutputRoot = InputRoot;
PROPAGATE TO TERMINAL 'Out1';
SET OutputRoot = InputRoot;
PROPAGATE TO LABEL 'ThirdCopy';

```

In the above example, the content of OutputRoot is reset before each PROPAGATE, because by default the node clears the output message buffer and reclaims the memory when the PROPAGATE statement completes. An alternative method is to instruct the node not to clear the output message on the first two PROPAGATE

statements, so that the message is available for routing to the next destination. The code to do this is:

```
SET OutputRoot = InputRoot;
PROPAGATE DELETE NONE;
SET OutputRoot = InputRoot;
PROPAGATE TO TERMINAL 'Out1' DELETE NONE;
SET OutputRoot = InputRoot;
PROPAGATE TO LABEL 'ThirdCopy';
```

If you do not initialize the output buffer, an empty message is generated, and the message flow detects an error and throws an exception.

Also ensure that you copy all required message headers to the output message buffer for each output message that you propagate.

If you want to modify the output message content before propagating each message, code the appropriate ESQL to make the changes that you want before you code the PROPAGATE statement.

If you set up the contents of the last output message that you want to generate, and propagate it as the final action of the Compute node, you do not have to include the final PROPAGATE statement. The default action of the Compute node is to propagate the contents of the output buffer when it terminates. This is implemented by the RETURN TRUE statement, included as the final statement in the module skeleton.

For example, if you want to generate three copies of the input message, and not perform any further action, include this code immediately before the RETURN TRUE statement:

```
SET OutputRoot = InputRoot;
PROPAGATE DELETE NONE;
PROPAGATE DELETE NONE;
```

Alternatively, you can modify the default behavior of the node by changing RETURN TRUE to RETURN FALSE:

```
SET OutputRoot = InputRoot;
PROPAGATE DELETE NONE;
PROPAGATE DELETE NONE;
PROPAGATE;
RETURN FALSE;
```

Three output messages are generated by the three PROPAGATE statements. The final RETURN FALSE statement causes the node to terminate but not propagate a final output message. Note that the final PROPAGATE statement does not include the DELETE NONE clause, because the node must release the memory at this stage.

Using numeric operators with datetime values:

This topic provides some examples of the ESQL that you can code to manipulate datetime values with numeric operators.

Adding an interval to a datetime value

The simplest operation you can perform is to add an interval to, or

subtract an interval from, a datetime value. For example, you could write the following expressions:

```
DATE '2000-03-29' + INTERVAL '1' MONTH  
TIMESTAMP '1999-12-31 23:59:59' + INTERVAL '1' SECOND
```

Adding or subtracting two intervals

Two interval values can be combined using addition or subtraction. The two interval values must be of compatible types. It is not valid to add a year-month interval to a day-second interval as in the following example:

```
INTERVAL '1-06' YEAR TO MONTH + INTERVAL '20' DAY
```

The interval qualifier of the resultant interval is sufficient to encompass all of the fields present in the two operand intervals. For example:

```
INTERVAL '2 01' DAY TO HOUR + INTERVAL '123:59' MINUTE TO SECOND
```

results in an interval with qualifier DAY TO SECOND, because both day and second fields are present in at least one of the operand values.

Subtracting two datetime values

Two datetime values can be subtracted to return an interval. In order to do this an interval qualifier must be given in the expression to indicate what precision the result should be returned in. For example:

```
(CURRENT_DATE - DATE '1776-07-04') DAY
```

returns the number of days since the 4th July 1776, whereas:

```
(CURRENT_TIME - TIME '00:00:00') MINUTE TO SECOND
```

returns the age of the day in minutes and seconds.

Scaling intervals

An interval value can be multiplied by or divided by an integer factor:

```
INTERVAL '2:30' MINUTE TO SECOND / 4
```

Calculating a time interval:

This ESQL example calculates the time interval between an input WebSphere MQ message being put on the input queue, and the time that it is processed in the current Compute node.

(When you make a call to a CURRENT_ datetime function, the value that is returned is identical to the value returned to another call in the same node. This ensures that you can use the function consistently within a single node.)

```
CALL CopyMessageHeaders();  
Declare PutTime INTERVAL;  
  
SET PutTime = (CURRENT_GMTIME - InputRoot.MQMD.PutTime) MINUTE TO SECOND;  
  
SET OutputRoot.XML.Test.PutTime = PutTime;
```

The output message has the format (although actual values vary):

```
<Test>  
<PutTime>INTERVAL '1:21.862' MINUTE TO SECOND</PutTime>  
</Test>
```


The following code snippet sets a timer, to be triggered after a specified interval from the start of processing, in order to check that processing has completed. If processing has not completed within the elapsed time, the firing of the timer might, for example, trigger some recovery processing.

The StartTime field of the timeout request message is set to the current time plus the allowed delay period, which is defined by a user-defined property on the flow. (The user-defined property has been set to a string of the form "HH:MM:SS" by the administrator.)

```

DECLARE StartDelayIntervalStr EXTERNAL CHARACTER '01:15:05';

CREATE PROCEDURE ValidateTimeoutRequest() BEGIN

    -- Set the timeout period
    DECLARE timeoutStartTimeRef REFERENCE TO
        OutputRoot.XMLNSC.Envelope.Header.TimeoutRequest.StartTime;
    IF LASTMOVE(timeoutStartTimeRef)
    THEN
        -- Already set
    ELSE
        -- Set it from the UDP StartDelayIntervalStr
        DECLARE startAtTime TIME CURRENT_TIME
            + CAST(StartDelayIntervalStr AS INTERVAL HOUR TO SECOND);

        -- Convert "TIME 'hh.mm.ss.fff'" to hh.mm.ss format
        -- needed in StartTime field
        DECLARE startAtTimeStr CHAR;
        SET startAtTimeStr = startAtTime;
        SET startAtTimeStr = SUBSTRING(startAtTimeStr FROM 7 FOR 8);
        SET OutputRoot.XMLNSC.Envelope.Header.TimeoutRequest.StartTime
            = startAtTimeStr;
    END IF;
END;

```

Selecting a subfield from a larger field:

You might have a message flow that processes a message containing delimited subfields. You can code ESQL to extract a subfield from the surrounding content if you know the delimiters of the subfield.

If you create a function that performs this task, or a similar one, you can invoke it both from ESQL modules (for Compute, Database, and Filter nodes) and from mapping files (used by DataDelete, DataInsert, DataUpdate, Extract, Mapping, and Warehouse nodes).

The following function example extracts a particular subfield of a message that is delimited by a specific character.

```

CREATE FUNCTION SelectSubField
    (SourceString CHAR, Delimiter CHAR, TargetStringPosition INT)
    RETURNS CHAR
-- This function returns a substring at parameter position TargetStringPosition within the
-- passed parameter SourceString. An example of use might be:
-- SelectSubField(MySourceField,' ',2) which will select the second subfield from the
-- field MySourceField delimited by a blank. If MySourceField has the value
-- "First Second Third" the function will return the value "Second"
BEGIN
    DECLARE DelimiterPosition INT;
    DECLARE CurrentFieldPosition INT 1;
    DECLARE StartNewString INT 1;
    DECLARE WorkingSource CHAR SourceString;
    SET DelimiterPosition = POSITION(Delimiter IN SourceString);
    WHILE CurrentFieldPosition < TargetStringPosition

```

```

DO
  IF DelimiterPosition = 0 THEN
  -- DelimiterPosition will be 0 if the delimiter is not found
  -- exit the loop
  SET CurrentFieldPosition = TargetStringPosition;
  ELSE
  SET StartNewString = DelimiterPosition + 1;
  SET WorkingSource = SUBSTRING(WorkingSource FROM StartNewString);
  SET DelimiterPosition = POSITION(Delimiter IN WorkingSource);
  SET CurrentFieldPosition = CurrentFieldPosition + 1;
  END IF;
END WHILE;
IF DelimiterPosition > 0 THEN
  -- Remove anything following the delimiter from the string
  SET WorkingSource = SUBSTRING(WorkingSource FROM 1 FOR DelimiterPosition);
  SET WorkingSource = TRIM(TRAILING Delimiter FROM WorkingSource);
END IF;
RETURN WorkingSource;
END;

```

Copying repeating fields:

You can configure a node with ESQL to copy repeating fields in several ways.

Consider an input XML message that contains a repeating structure:

```

...
<Field_top>
  <field1></field1>
  <field1></field1>
  <field1></field1>
  <field1></field1>
  <field1></field1>
</Field_top>
.....

```

You cannot copy this whole structure field with the following statement:

```
SET OutputRoot.XML.Output_top.Outfield1 = InputRoot.XML.Field_top.field1;
```

That statement copies only the first repeat, and therefore produces the same result as this statement:

```
SET OutputRoot.XML.Output_top.Outfield1[1] = InputRoot.XML.Field_top.field1[1];
```

You can copy the fields within a loop, controlling the iterations with the **CARDINALITY** of the input field:

```

SET I = 1;
SET J = CARDINALITY(InputRoot.XML.Field_top.field1[]);
WHILE I <= J DO
  SET OutputRoot.XML.Output_top.Outfield1[I] = InputRoot.XML.Field_top.field1[I];
  SET I = I + 1;
END WHILE;

```

This might be appropriate if you want to modify each field in the output message as you copy it from the input field (for example, add a number to it, or fold its contents to uppercase), or after it has been copied. If the output message already contained more Field1 fields than existed in the input message, the surplus fields would not be modified by the loop and would remain in the output message.

The following single statement copies the iterations of the input fields to the output fields, and deletes any surplus fields in the output message.

```
SET OutputRoot.XML.Output_top.Outfield1.[] = InputRoot.XML.Field_top.field1[];
```

The example below shows how you can rename the elements when you copy them into the output tree. This statement does not copy across the source element name, therefore each field1 element becomes a Target element.

```
SET OutputRoot.XML.Output_top.Outfield1.Target[] =
  (SELECT I FROM InputRoot.XML.Field_top.field1[] AS I );
```

The next example shows a different way to do the same operation; it produces the same end result.

```
SET OutputRoot.XML.Output_top.Outfield2.Target[]
  = InputRoot.XML.Field_top.field1[];
```

The following example copies across the source element name. Each field1 element is retained as a field1 element under the Target element.

```
SET OutputRoot.XML.Output_top.Outfield3.Target.[]
  = InputRoot.XML.Field_top.field1[];
```

This example is an alternative way to achieve the same result, with field1 elements created under the Target element.

```
SET OutputRoot.XML.Output_top.Outfield4.Target.*[]
  = InputRoot.XML.Field_top.field1[];
```

These examples show that there are several ways in which you can code ESQL to copy repeating fields from source to target. Select the most appropriate method to achieve the results that you require.

The principals shown here apply equally to all areas of the message tree to which you can write data, not just the output message tree.

A note about copying fields:

Be aware that, when copying an input message element to an output element, not only the *value* of the output element but also its *type* is set to that of the input element. This means that if, for example, you have an input XML document with an attribute, and you want to set a Field element (rather than an attribute) in your output message to the value of the input attribute, you have to include a TYPE clause cast to change the element-type from attribute to Field.

For example, given an input like:

```
<Field01 Attrib01='Attrib01_Value'>Field01_Value</Field01>
```

To create an output like:

```
<MyField_A MyAttrib_A='Attrib01_Value' MyAttrib_B='Field01_Value' >
  <MyField_B>Field01_Value</MyField_BC>
  <MyField_C>Attrib01_Value</MyField_C>
</MyField_A'>
```

You would use the following ESQL:

```
-- Create output attribute from input attribute
SET OutputRoot.XMLNSC.MyField_A.MyAttrib_A = InputRoot.XMLNSC.Field01.Attrib01;
-- Create output field from input field
SET OutputRoot.XMLNSC.MyField_A.MyField_B = InputRoot.XMLNSC.Field01;

-- Create output attribute from input field value, noting we have to
-- "cast" back to an attribute element
SET OutputRoot.XMLNSC.MyField_A.(XMLNSC.Attribute)MyAttrib_B =
  InputRoot.XMLNSC.Field01;

-- Create output field from input attribute value, noting we have to
```

```
-- "cast" back to a field element
SET OutputRoot.XMLNSC.MyField_A.(XMLNSC.Field)MyField_C =
    InputRoot.XMLNSC.Field01.Attrib01;
```

Manipulating repeating fields in a message tree:

This topic describes the use of the SELECT function, and other column functions, to manipulate repeating fields in a message tree.

Suppose that you want to perform a special action on invoices that have a total order value greater than a certain amount. To calculate the total order value of an Invoice field, you must multiply the Price fields by the Quantity fields in all the Items in the message, and total the result. You can do this using a SELECT expression as follows:

```
(
  SELECT SUM( CAST(I.Price AS DECIMAL) * CAST(I.Quantity AS INTEGER) )
    FROM Body.Invoice.Purchases."Item"[] AS I
)
```

The example assumes that you need to use CAST expressions to cast the string values of the fields Price and Quantity into the correct data types. The cast of the Price field into a decimal produces a decimal value with the *natural* scale and precision, that is, whatever scale and precision is necessary to represent the number. These CASTs would not be necessary if the data were already in an appropriate data type.

The SELECT expression works in a similar way to the quantified predicate, and in much the same way that a SELECT works in standard database SQL. The FROM clause specifies what is being iterated, in this case, all Item fields in Invoice, and establishes that the current instance of Item can be referred to using I. This form of SELECT involves a column function, in this case the SUM function, so the SELECT is evaluated by adding together the results of evaluating the expression inside the SUM function for each Item field in the Invoice. As with standard SQL, NULL values are ignored by column functions, with the exception of the COUNT column function explained below, and a NULL value is returned by the column function only if there are no non-NULL values to combine.

The other column functions that are provided are MAX, MIN, and COUNT. The COUNT function has two forms that work in different ways with regard to NULLs. In the first form you use it much like the SUM function above, for example:

```
SELECT COUNT(I.Quantity)
  FROM Body.Invoice.Purchases."Item"[] AS I
```

This expression returns the number of Item fields for which the Quantity field is non-NULL. That is, the COUNT function counts non-NULL values, in the same way that the SUM function adds non-NULL values. The alternative way of using the COUNT function is as follows:

```
SELECT COUNT(*)
  FROM Body.Invoice.Purchases."Item"[] AS I
```

Using COUNT(*) counts the total number of Item fields, regardless of whether any of the fields is NULL. The above example is in fact equivalent to using the CARDINALITY function, as in the following:

CARDINALITY(Body.Invoice.Purchases."Item" [])

In all the examples of SELECT given here, just as in standard SQL, you could use a WHERE clause to provide filtering on the fields.

Manipulating other parts of the message tree

The following topics describe how you can access parts of the message tree other than the message body data. These parts of the logical tree are independent of the domain in which the message exists, and all these topics apply to messages in the BLOB domain in addition to all other supported domains.

- “Accessing headers”
- “Accessing the Properties tree” on page 190
- “Accessing the LocalEnvironment tree” on page 192
- “Accessing the Environment tree” on page 195
- “Accessing the ExceptionList tree” on page 196

Accessing headers:

If the input message received by an input node includes message headers that are recognized by the input node, the node invokes the correct parser for each header. Parsers are supplied for most WebSphere MQ headers. The topics listed below provide some guidance for accessing the information in the MQMD and MQRFH2 headers, which you can follow as general guidance for accessing other headers also present in your messages.

Every header has its own correlation name, for example, MQMD, and you must use this in all ESQL statements that refer to or set the content of this tree:

- “Accessing the MQMD header”
- “Accessing the MQRFH2 header” on page 190

For further details of the contents of these and other WebSphere MQ headers for which WebSphere Message Broker provides a parser, see “Element definitions for message parsers” on page 735.

Accessing the MQMD header:

WebSphere MQ, WebSphere MQ Everyplace, and SCADA messages include an MQMD header. You can refer to the fields within the MQMD, and you can update them in a Compute node.

For example, you might want to copy the message identifier MSGID in the MQMD to another field in your output message. To do that, code:

```
SET OutputRoot.MRM.Identifier = InputRoot.MQMD.MsgId;
```

If you send a message to an EBCDIC system from a distributed system, you might need to convert the message to a compatible CodedCharSetId and Encoding. To do this, code the following ESQL in the Compute node:

```
SET OutputRoot.MQMD.CodedCharSetId = 500;  
SET OutputRoot.MQMD.Encoding = 785;
```

The MQMD properties of CodedCharSetId and Encoding define the code page and encoding of the section of the message that follows (typically this is either the MQRFH2 header or the message body itself).

Accessing the MQRFH2 header:

When you construct MQRFH2 headers in a Compute node, there are two types of fields:

- Fields in the MQRFH2 header structure (for example, Format and NameValueCCSID)
- Fields in the MQRFH2 NameValue buffer (for example mcd and psc)

To differentiate between these two field types, insert a value in front of the referenced field in the MQRFH2 field to identify its type (a value for the NameValue buffer is not required because this is the default). The value that you specify for the header structure is (MQRFH2.Field).

For example:

- To create or change an MQRFH2 Format field, specify the following ESQL:

```
SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR  ';
```

- To create or change the psc folder with a topic:

```
SET OutputRoot.MQRFH2.psc.Topic = 'department';
```

- To add an MQRFH2 header to an outgoing message that is to be used to make a subscription request:

```
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);

WHILE I < J DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;

SET OutputRoot.MQRFH2.(MQRFH2.Field)Version = 2;
SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR';
SET OutputRoot.MQRFH2.(MQRFH2.Field)NameValueCCSID = 1208;
SET OutputRoot.MQRFH2.psc.Command = 'RegSub';
SET OutputRoot.MQRFH2.psc.Topic = "InputRoot"."MRM"."tope1";
SET OutputRoot.MQRFH2.psc.QMgrName = 'DebugQM';
SET OutputRoot.MQRFH2.psc.QName = 'PUBOUT';
SET OutputRoot.MQRFH2.psc.RegOpt = 'PersAsPub';
```

Note the use of a variable, J, initialized to the value of the cardinality of the existing headers in the message. This is more efficient than calculating the cardinality on each iteration of the loop, which happens if you code the following WHILE statement:

```
WHILE I < CARDINALITY(InputRoot.*[]) DO
```

Note: The MQRFH2 header can be parsed in either the MQRFH2 parser domain or the MQRFH2C compact parser domain. Use the MQRFH2C compact parser by selecting the Use MQRFH2C Compact Parser for MQRFH2 Domain check box on the input node of the message flow.

Accessing the Properties tree:

The Properties tree has its own correlation name, Properties, and you must use this in all ESQL statements that refer to or set the content of this tree.

The fields in the Properties tree contain values that define the characteristics of the message. For example, the Properties tree contains a field for the message domain, and fields for the encoding and CCSID in which message data is encoded. For a full list of fields in this tree, see “Data types for elements in the Properties subtree” on page 736.

You can interrogate and update these fields using the appropriate ESQL statements. If you create a new output message in the Compute node, you must set values for the message properties.

Setting output message properties:

If you use the Compute node to generate a new output message, you must set its properties in the Properties tree. The output message properties do not have to be the same as the input message properties.

For example, to set the output message properties for an output MRM message, set the following properties:

Property	Value
Message Domain	MRM
Message Set	Message set identifier
Message Type	Message name ¹
Message Format	Physical format name ²

Notes:

1. If you are using multipart messages, refer to Multipart messages for details of how *MessageType* is used.
2. The name that you specify for the physical layer must match the name that you have defined for it. The default physical layer names are CWF1, XML1, and TDS1.

This ESQL procedure sets message properties to values passed in by the calling statement. You might find that you have to perform this task frequently, and you can use a procedure like this in many different nodes and message flows. If you prefer, you can code ESQL that sets specific values.

```
CREATE PROCEDURE setMessageProperties(IN OutputRoot REFERENCE, IN setName char,
                                   IN typeName char, IN formatName char) BEGIN
  /*****
  * A procedure that sets the message properties
  *****/
  set OutputRoot.Properties.MessageSet = setName;
  set OutputRoot.Properties.MessageType = typeName;
  set OutputRoot.Properties.MessageFormat = formatName;
END;
```

To set the output message domain, you can set the message property, or you can code ESQL statements that refer to the required domain in the second qualifier of the SET statement, the parser field. For example, the ESQL statement sets the domain to MRM:

```
SET OutputRoot.MRM.Field1 = 'field1 data';
```

This ESQL statement sets the domain to XML:

```
SET OutputRoot.XML.Field1 = 'field1 data';
```

Do not specify more than one domain in the ESQL for any single message. However, if you use PROPAGATE statements to generate several output messages, you can set a different domain for each message.

For information about the full list of elements in the Properties tree, see “Data types for elements in the Properties subtree” on page 736.

Accessing the LocalEnvironment tree:

The LocalEnvironment tree has its own correlation name, LocalEnvironment, and you must use this in all ESQL statements that refer to or set the content of this tree.

The LocalEnvironment tree is used by the broker, and you can refer to and modify this information. You can also extend the tree to contain information that you create yourself. You can create subtrees within this tree that you can use as a scratchpad or working area.

The message flow sets up information in two subtrees, Destination and WrittenDestination, below the LocalEnvironment root. You can refer to the content of both of these, and can write to them to influence the way in which the message flow processes your message. However, if you write to these areas, ensure that you follow the defined structure to ensure that the tree remains valid.

If you want the LocalEnvironment tree to be included in the output message that is propagated by the Compute node, you must set the Compute node property *Compute mode* to a value that includes LocalEnvironment (for example, All). If you do not, the LocalEnvironment tree is not copied to the output message.

The information that you insert into DestinationData or Defaults depends on the characteristic of the corresponding node property:

- If a node property is represented by a check box (for example, *New Message ID*), set the Defaults or DestinationData element to Yes (equivalent to selecting the check box) or No (equivalent to clearing the check box).
- If a node property is represented by a drop-down list (for example, *Transaction Mode*), set the Defaults or DestinationData element to the appropriate character string (for example Automatic).
- If a node property is represented by a text entry field (for example, *Queue Manager Name*), set the Defaults or DestinationData element to the character string that you would enter in this field.

If necessary, configure the sending node to indicate where the destination information is. For example, for the output node MQOutput, set *Destination Mode*:

- If you set *Destination Mode* to Queue Name, the output message is sent to the queue identified in the output node properties *Queue Name* and *Queue Manager Name*. Destination is not referenced by the node.
- If you set *Destination Mode* to Destination List, the node extracts the destination information from the Destination subtree. This means that you can send a single message to multiple destinations, if you configure Destination and

a single output node correctly. The node only checks the node properties if a value is not available in Destination (as described above).

- If you set *Destination Mode* to Reply To Queue, the message is sent to the reply-to queue identified in the MQMD in this message (field ReplyToQ). Destination is not referenced by the node.

“Populating Destination in the LocalEnvironment tree” on page 194 includes ESQL procedures that perform typical updates to the LocalEnvironment. Review the ESQL statements in these procedures to see how to modify LocalEnvironment. You can use these procedures unchanged, or modify them for your own requirements.

“Using scratchpad areas in LocalEnvironment” describes how to extend the contents of this tree for your own purposes.

For another example of how you can use LocalEnvironment to modify the behavior of a message flow, refer to the XML_PassengerQuery message flow in the Airline Reservations sample program. The Compute node in this message flow writes a list of destinations in the RouterList subtree of Destination that are used as labels by a later RouteToLabel node that propagates the message to the corresponding Label node.

Using scratchpad areas in LocalEnvironment:

The LocalEnvironment tree includes a subtree called Variables. This is always created, but is never populated by the message flow. Use this area for your own purposes, for example to pass information from one node to another. You can create other subtrees in the LocalEnvironment tree if you choose.

The advantage of creating your own data in a scratchpad in the LocalEnvironment is that this data can be propagated as part of the logical tree to subsequent nodes in the message flow. If you create a new output message in a Compute node, you can also include all or part of the LocalEnvironment tree from the input message in the new output message. If you want to do this, you must set the *Compute mode* property of the Compute node to include LocalEnvironment as part of the output tree (for example, specify A11). (You also include the ExceptionList tree in your output message. See the “Compute node” on page 492 for further details about *Compute mode*.)

However, any data updates or additions that you make in one node are not retained if the message flows backwards through the message flow (for example, if an exception is thrown, or if the message is processed through the second terminal of the FlowOrder node). If you create your own data, and want that data to be preserved throughout the message flow, you must use the Environment tree.

You can set values in the Variables subtree in a Compute node that are used later by another node (Compute, Database, or Filter) for some purpose that you determine when you configure the message flow.

For example, you might use this to determine the destination of an output message. Your first Compute node could determine in some way that the output messages from this message flow must go to WebSphere MQ queues. Include the following ESQL to insert this information into the LocalEnvironment scratchpad area in the message that the Compute node sends to the next node in the message flow:

```
SET OutputLocalEnvironment.Variables.OutputLocation = 'MQ';
```

Your second Compute node can access this information from its input message. In the ESQL in this node, use the correlation name `InputLocalEnvironment` to identify the `LocalEnvironment` tree within the input message that contains this data. Set the *Compute mode* to include the `LocalEnvironment` tree in the output message and copy the data from the `InputLocalEnvironment` to the `Destination` subtree in the output message. Configure the `MQOutput` node to use the destination list that you have created in the `LocalEnvironment` tree by setting property *Destination Mode* to `Destination List`.

For information about the full list of elements in the `DestinationData` subtree, see “Data types for elements in the `DestinationData` subtree” on page 736.

Populating Destination in the LocalEnvironment tree:

You can use the `Destination` subtree to set up target destinations that are used by output nodes, the `HTTPRequest` node, and the `RouteToLabel` node. The examples below show how you can create and use an ESQL procedure to perform the task of setting up values for each of these uses.

You can copy and use these procedures as shown, or you can modify or extend them to perform similar tasks.

Adding a queue name for the MQOutput node.

```
| CREATE PROCEDURE addToMQDestinationList(IN LocalEnvironment REFERENCE, IN newQueue char) BEGIN
| /*****
| * A procedure that will add a queue name to the MQ destination list
| * in the local environment.
| * This list is used by a MQOutput node which has its mode set to Destination list.
| *
| * IN LocalEnvironment: LocalEnvironment to be modified.
| * Set this to OutputLocalEnvironment when calling this procedure
| * IN queue:    queue to be added to the list
| *
| *****/
| DECLARE I INTEGER CARDINALITY(LocalEnvironment.Destination.MQDestinationList.DestinationData[]);
| IF I = 0 THEN
|   SET LocalEnvironment.Destination.MQDestinationList.DestinationData[1].queueName = newQueue;
| ELSE
|   SET LocalEnvironment.Destination.MQDestinationList.DestinationData[I+1].queueName = newQueue;
| END IF;
| END;
```

Changing the default URL for an HTTPRequest node request.

```
CREATE PROCEDURE overrideDefaultHTTPRequestURL(IN LocalEnvironment REFERENCE, IN newUrl char) BEGIN
/*****
* A procedure that will change the URL to which the HTTPRequest node will send the request.
*
* IN LocalEnvironment: LocalEnvironment to be modified.
* Set this to OutputLocalEnvironment when calling this procedure
* IN queue:    URL to send the request to.
*
*****/
set LocalEnvironment.Destination.HTTP.RequestURL = newUrl;
END;
```

Adding a label for the RouteToLabel node.

```

CREATE PROCEDURE addToRouteToLabelList(IN LocalEnvironment REFERENCE, IN newLabel char) BEGIN
  /*****
  * A procedure that will add a label name to the RouteToLabel list
  * in the local environment.
  * This list is used by a RoteToLabel node.
  *
  * IN LocalEnvironment: LocalEnvironment to be modified.
  * Set this to OutputLocalEnvironment when calling this procedure
  * IN label: label to be added to the list
  *
  *****/
  if LocalEnvironment.Destination.RouterList.DestinationData is null then
    set LocalEnvironment.Destination.RouterList.DestinationData."label" = newLabel;
  else
    create LASTCHILD OF LocalEnvironment.Destination.RouterList.DestinationData
    NAME 'label' VALUE newLabel;
  end if;
END;

```

Accessing the Environment tree:

The Environment tree has its own correlation name, Environment, and you must use this in all ESQL statements that refer to or set the content of this tree.

The Environment tree is always created when the logical tree is created for an input message. However the message flow neither populates it nor uses its contents. You can use this tree for your own purposes, for example to pass information from one node to another. You can use the whole tree as a scratchpad or working area.

The advantage of creating your own data in Environment is that this data is propagated as part of the logical tree to subsequent nodes in the message flow. If you create a new output message in a Compute node, the Environment tree is also copied from the input message to the new output message. (This is in contrast to the LocalEnvironment tree, which is only included in the output message if you explicitly request that it is).

Only one Environment tree is present for the duration of the message flow. Any data updates or additions that you make in one node are retained and all nodes in the message flow have access to the latest copy of this tree. Even if the message flows back through the message flow (for example, if an exception is thrown, or if the message is processed through the second terminal of the FlowOrder node), the latest state is retained.

This is in contrast to the LocalEnvironment tree, which reverts to its previous state if the message flows back through the message flow.

You can use this tree for any purpose you choose. For example, you could use the following ESQL statements to create fields in the tree:

```

SET Environment.Variables =
  ROW('granary' AS bread, 'reisling' AS wine, 'stilton' AS cheese);
SET Environment.Variables.Colors[] =
  LIST{'yellow', 'green', 'blue', 'red', 'black'};
SET Environment.Variables.Country[] = LIST{ROW('UK' AS name, 'pound' AS currency),
  ROW('USA' AS name, 'dollar' AS currency)};

```

This information is now available to all nodes to which a message is propagated, regardless of their relative position in the message flow.

For another example of how you can use Environment to store information used by other nodes in the message flow, refer to the Reservation message flow in the Airline sample program. The Compute node in this message flow writes information to the subtree Environment.Variables that it has extracted from a database according to the value of a field in the input message.

Accessing the ExceptionList tree:

The ExceptionList tree has its own correlation name, ExceptionList, and you must use this in all ESQL statements that refer to or set the content of this tree.

This tree is created with the logical tree when an input message is parsed. It is initially empty, and is only populated if an exception occurs during message flow processing. It is possible that more than one exception can occur; if this happens, the ExceptionList tree contains a subtree for each exception.

You can access the ExceptionList tree in Compute, Database, and Filter nodes, and you can update it in a Compute node. You must use the appropriate correlation name; Exception List for a Database or Filter node, and InputExceptionList for a Compute node.

You might want to access this tree in a node in an error handling procedure. For example, you might want to route the message to a different path based on the type of exception, for example one that you have explicitly generated using an ESQL THROW statement, or one that the broker has generated.

The following ESQL shows how you can access the ExceptionList and process each child that it contains:

```
-- Declare a reference for the ExceptionList
-- (in a Compute node use InputExceptionList)
DECLARE start REFERENCE TO ExceptionList.*[1];

-- Loop through the exception list children
WHILE start.Number IS NOT NULL DO
  -- more ESQL

  -- Move start to the last child of the field to which it currently points
  MOVE start LASTCHILD;
END WHILE;
```

The second example below shows an extract of ESQL that has been coded for a Compute node to loop through the exception list to the last (nested) exception description and extract the error number. This error relates to the original cause of the problem and normally provides the most precise information. Subsequent action taken by the message flow can be decided by the error number retrieved in this way.

```

CREATE PROCEDURE getLastExceptionDetail(IN InputTree reference,OUT messageNumber integer,
OUT messageText char)
  /*****
  * A procedure that will get the details of the last exception from a message
  * IN InputTree: The incoming exception list
  * IN messageNumber: The last message numberr.
  * IN messageText: The last message text.
  *****/
BEGIN
  -- Create a reference to the first child of the exception list
  declare ptrException reference to InputTree.*[1];
  -- keep looping while the moves to the child of exception list work
WHILE lastmove(ptrException) DO
  -- store the current values for the error number and text
  IF ptrException.Number is not null THEN
    SET messageNumber = ptrException.Number;
    SET messageText = ptrException.Text;
  END IF;
  -- now move to the last child which should be the next exceptionlist
  move ptrException lastchild;
END WHILE;
END;

```

For more information about the use of ExceptionList, refer to the subflow in the Error Handler sample, which includes ESQL that interrogates the ExceptionList structure and takes specific action according to its content.

Transforming from one data type to another

You can use ESQL to transform messages and data types in many ways. The following topics provide guidance about the following:

- “Casting data from message fields”
- “Converting code page and message encoding” on page 198
- “Converting EBCDIC NL to ASCII CR LF” on page 200
- “Changing message format” on page 202

Casting data from message fields:

When you compare an element with another element, variable or constant, ensure that the value with which you are comparing the element is consistent (for example, character with character). If the values are not consistent, the broker generates a runtime error if it cannot provide an implicit casting to resolve the inconsistency. For details of what implicit casts are supported, see “Implicit casts” on page 973.

You can use the CAST function to transform the data type of one value to match the data type of the other. For example, you can use the CAST function when you process generic XML messages. All fields in an XML message have character values, so if you want to perform arithmetic calculations or datetime comparisons, for example, you must convert the string value of the field into a value of the appropriate type using CAST.

In the Invoice message, the field InvoiceDate contains the date of the invoice. If you want to refer to or manipulate this field, you must CAST it to the correct format first. For example, to refer to this field in a test:

```
IF CAST(Body.Invoice.InvoiceDate AS DATE) = CURRENT_DATE THEN
```

This converts the string value of the InvoiceDate field into a date value, and compares it to the current date.

Another example is casting from integer to character:

```
DECLARE I INTEGER 1;
DECLARE C CHARACTER;

-- The following statement generates an error
SET C = I;

-- The following statement is valid
SET C = CAST(I AS CHARACTER);
```

Converting code page and message encoding:

You can use ESQL within a Compute node to convert data for code page and message encoding. If your message flow is processing WebSphere MQ messages, you can use WebSphere MQ facilities (including get and put options and WebSphere MQ data conversion exits) to provide these conversions. If you are not processing WebSphere MQ messages, or you choose not to use WebSphere MQ facilities, you can use WebSphere Message Broker facilities by coding the appropriate ESQL in a Compute node in your message flow.

The contents of the MQMD, the MQRFH2, and the message body of a message in the MRM domain that has been modeled with a CWF physical format can be subject to code page and encoding conversion. The contents of a message body of a message in the XML, XMLNS, and JMS domains, and those messages in the MRM domain that have been modeled with an XML or TDS physical format, are treated as strings. Only code page conversion applies; no encoding conversion is required.

For messages in the MRM domain modeled with a CWF physical format, you can set the MQMD CCSID and Encoding fields of the output message, plus the CCSID and Encoding of any additional headers, to the required target value.

For messages in the MRM domain modeled with an XML or TDS physical format, you can set the MQMD CCSID field of the output message, plus the CCSID of any additional headers. XML and TDS data is handled as strings and is therefore subject to CCSID conversion only.

An example WebSphere MQ message has an MQMD header, an MQRFH2 header, and a message body. To convert this message to a mainframe CodedCharSetId and Encoding, code the following ESQL in the Compute node:

```
SET OutputRoot.MQMD.CodedCharSetId = 500;
SET OutputRoot.MQMD.Encoding = 785;
SET OutputRoot.MQRFH2.CodedCharSetId = 500;
SET OutputRoot.MQRFH2.Encoding = 785;
```

The following example illustrates what you must do to modify a CWF message so that it can be passed from WebSphere Message Broker to IMS[™] on z/OS.

1. You have defined the input message in XML and are using an MQRFH2 header. Remove the header before passing the message to IMS.
2. The message passed to IMS must have MQIIH header, and must be in the z/OS code page. This message is modeled in the MRM and has the name IMS1. Define the PIC X fields in this message as logical type string for

conversions between EBCDIC and ASCII to take place. If they are logical type binary, no data conversion occurs; binary data is ignored when a CWF message is parsed by the MRM parser.

3. The message received from IMS is also defined in the MRM and has the name IMS2. Define the PIC X fields in this message as logical type string for conversions between EBCDIC and ASCII to take place. If they are logical type binary, no data conversion occurs; binary data is ignored when a CWF message is parsed by the MRM parser.
4. Convert the reply message to the Windows code page. The MQIIH header is retained on this message.
5. You have created a message flow that contains the following nodes :
 - a. The outbound flow, **MQInput1 --> Compute1 --> MQOutput1**.
 - b. The inbound flow, **MQInput2 --> Compute2 --> MQOutput2**.
6. Code ESQL in Compute1 (outbound) node as follows, specifying the relevant MessageSet id. This code shows the use of the default CWF physical layer name. You must use the name that matches your model definitions. If you specify an incorrect value, the broker fails with message BIP5431.

```
-- Loop to copy message headers
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);

WHILE I < J - 1 DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;

SET OutputRoot.MQMD.CodedCharSetId = 500;
SET OutputRoot.MQMD.Encoding = 785;
SET OutputRoot.MQMD.Format = 'MQIMS  ';
SET OutputRoot.MQIIH.StrucId = 'IIH  ';
SET OutputRoot.MQIIH.Version = 1;
SET OutputRoot.MQIIH.StrucLength = 84;
SET OutputRoot.MQIIH.Encoding = 785;
SET OutputRoot.MQIIH.CodedCharSetId = 500;
SET OutputRoot.MQIIH.Format = 'MQIMSVS  ';
SET OutputRoot.MQIIH.Flags = 0;
SET OutputRoot.MQIIH.LTermOverride = '      ';
SET OutputRoot.MQIIH.MFSMapName = '      ';
SET OutputRoot.MQIIH.ReplyToFormat = 'MQIMSVS  ';
SET OutputRoot.MQIIH.Authenticator = '      ';
SET OutputRoot.MQIIH.TranInstanceId = X'00000000000000000000000000000000';
SET OutputRoot.MQIIH.TranState = '  ';
SET OutputRoot.MQIIH.CommitMode = '0';
SET OutputRoot.MQIIH.SecurityScope = 'C';
SET OutputRoot.MQIIH.Reserved = '  ';
SET OutputRoot.MRM.e_e1en08 = 30;
SET OutputRoot.MRM.e_e1en09 = 0;
SET OutputRoot.MRM.e_string08 = InputBody.e_string01;
SET OutputRoot.MRM.e_binary02 = X'31323334353637383940';
SET OutputRoot.Properties.MessageDomain = 'MRM';
SET OutputRoot.Properties.MessageSet = 'DHCJOEG072001';
SET OutputRoot.Properties.MessageType = 'IMS1';
SET OutputRoot.Properties.MessageFormat = 'CWF1';
```

Note the use of a variable, J, that is initialized to the value of the cardinality of the existing headers in the message. This is more efficient than calculating the cardinality on each iteration of the loop, which happens if you code the following WHILE statement:

```
WHILE I < CARDINALITY(InputRoot.*[]) DO
```

7. Create ESQL in Compute2 (inbound) node as follows, specifying the relevant MessageSet id. This code shows the use of the default CWF physical layer name. You must use the name that matches your model definition. If you specify an incorrect value, the broker fails with message BIP5431.

```
-- Loop to copy message headers
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);

WHILE I < J DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;

SET OutputRoot.MQMD.CodedCharSetId = 437;
SET OutputRoot.MQMD.Encoding = 546;
SET OutputRoot.MQMD.Format = 'MQIMS  ';
SET OutputRoot.MQIIH.CodedCharSetId = 437;
SET OutputRoot.MQIIH.Encoding = 546;
SET OutputRoot.MQIIH.Format = '      ';
SET OutputRoot.MRM = InputBody;
SET OutputRoot.Properties.MessageDomain = 'MRM';
SET OutputRoot.Properties.MessageSet = 'DHCJ0EG072001';
SET OutputRoot.Properties.MessageType = 'IMS2';
SET OutputRoot.Properties.MessageFormat = 'CWF1';
```

You do not have to set any specific values for the MQInput1 node properties, because the message and message set are identified in the MQRFH2 header, and no conversion is required.

You must set values for message domain, set, type, and format in the MQInput node for the inbound message flow (MQInput2). You do not need to set conversion parameters.

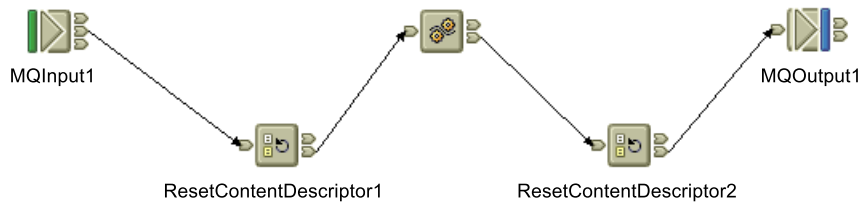
One specific situation in which you might need to convert data in one code page to another is when messages contain new line indicators and are passing between EBCDIC and ASCII systems. The required conversion for this situation is described in “Converting EBCDIC NL to ASCII CR LF.”

Converting EBCDIC NL to ASCII CR LF:

This topic describes an example task that changes new line (NL) characters in a text message to carriage return (CR) and line feed (LF) character pairs.

This conversion might be useful if messages from an EBCDIC platform (for example, using CCSID 1047) are sent to an ASCII platform (for example, using CCSID 437). Problems can arise because the EBCDIC NL character hex '15' is converted to the undefined ASCII character hex '7F'. There is no corresponding code point for the NL character in the ASCII code page.

In this example, a message flow is created that interprets the input message as a message in the BLOB domain. This is passed into a ResetContentDescriptor node to reset the data to a message in the MRM domain. The message is called msg_nl (a set of repeating string elements delimited by EBCDIC NL characters). A Compute node is then used to create an output based on another message in the MRM domain called msg_crlf (a set of repeating string elements delimited by CR LF pairs). The message domain is then changed back to BLOB in another ResetContentDescriptor node. This message flow is illustrated below.



The following instructions show how to create the messages and configure the message flow.

1. Create the message models for the messages in the MRM domain:
 - a. Create a message set project called myProj.
 - b. Create a message set called myMessageSet with a TDS physical format (the default name is TDS1).
 - c. Create an element string1 of type xsd:string.
 - d. Create a complex type called t_msg_nl and specify the following complex type properties:
 - *Composition* = Ordered Set
 - *Content Validation* = Closed
 - *Data Element Separation* = All Elements Delimited
 - *Delimiter* = <U+0085> (hex '0085' is the UTF-16 representation of a NL character)
 - *Repeat* = Yes
 - *Min Occurs* = 1
 - *Max Occurs* = 50 (the text of the message is assumed to consist of no more than 50 lines)
 - e. Add Element string1 and set the following property:
 - *Repeating Element Delimiter* = <U+0085>
 - f. Create a Message msg_nl and set its associated complex type to t_msg_nl
 - g. Create a complex type called t_msg_crlf and specify the following complex type properties:
 - *Composition* = Ordered Set
 - *Content Validation* = Closed
 - *Data Element Separation* = All Elements Delimited
 - *Delimiter* <CR><LF> (<CR> and <LF> are the mnemonics for the CR and LF characters)
 - *Repeat* = Yes
 - *Min Occurs* = 1
 - *Max Occurs* = 50
 - h. Add Element string1 and set the following property:
 - *Repeating Element Delimiter* = <CR><LF>
 - i. Create a Message msg_crlf and set complex type to t_msg_crlf.
2. Configure the message flow shown in the figure above:
 - a. Start with the MQInput1 node:
 - Set *Message Domain* = BLOB
 - Set *Queue Name* = <Your input message queue name>
 - b. Add the ResetContentDescriptor1 node, connected to the out terminal of MQInput1:

- Set *Message Domain* = MRM
 - Select *Reset Message Domain*
 - Set *Message Set* = <Your Message Set ID> (this has a maximum of 13 characters)
 - Select *Reset Message Set*
 - Set *Message Type* = msg_n1
 - Select *Reset Message Type*
 - Set *Message Format* = TDS1
 - Select *Reset Message Format*
- c. Add the Compute1 node, connected to the out terminal of ResetContentDescriptor1:
- Enter a name for the *ESQL Module* for this node, or accept the default (<message flow name>_Compute1).
 - Right-click the Compute1 node and select **Open ESQL**. Code the following ESQL in the module:

```
-- Declare local working variables
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);

-- Loop to copy all message headers from input to output message
WHILE I < J DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;

-- Set new output message type which uses CRLF delimiter
SET OutputRoot.Properties.MessageType = 't_msg_crlf';

-- Loop to copy each instance of string1 child within message body
SET I = 1;
SET J = CARDINALITY("InputBody"."string1"[]);
WHILE I <= J DO
  SET "OutputRoot"."MRM"."string1"[I] = "InputBody"."string1"[I];
  SET I=I+1;
END WHILE;
```

Note the use of a variable, J, initialized to the value of the cardinality of the existing headers in the message. This is more efficient than calculating the cardinality on each iteration of the loop, which happens if you code the following WHILE statement:

```
WHILE I < CARDINALITY(InputRoot.*[]) DO
```

- d. Add the ResetContentDescriptor2 node, connected to the out terminal of the Compute1 node:
- Set *Message Domain* = BLOB
 - Select *Reset Message Domain*.
- e. Finally, add the MQOutput1 node, connected to the out terminal of the ResetContentDescriptor2 node. Configure its properties to direct the output message to the required queue or queues.

Changing message format:

Use the Compute node to copy part of an input message to an output message. The results of such a copy depend on the type of input and output parsers involved.

Like parsers:

Where both the source and target messages have the same folder structure at root level, a **like-parser-copy** is performed. For example:

```
SET OutputRoot.MQMD = InputRoot.MQMD;
```

copies all the children in the MQMD folder of the input message to the MQMD folder of the output message.

Another example of a tree structure that supports a like-parser-copy is:

```
SET OutputRoot.XML.Data.Account = InputRoot.XML.Customer.Bank.Data;
```

If you want to transform an input message in the MRM domain to an output message also in the MRM domain, you can use either the Compute or the Mapping node. The Mapping node can interpret the action that is required because it knows the format of both messages. Content Assist in the ESQL module for the Compute node can also use the message definitions for those messages. If the messages are not in the same namespace, you must use the Compute node.

Note: To get Content Assist to work with message references, you must set up a project reference from the project containing the ESQL to the project containing the message set. For information about setting up a project reference, see Project references.

If both input and output messages are not in the MRM domain, you must use the Compute node and specify the structure of the messages yourself.

Unlike parsers:

Where the source and target messages have different folder structures at root level, you cannot make an exact copy of the message source. Instead, the **unlike-parser-copy** views the source message as a set of nested folders terminated by a leaf name-value pair. For example, copying the following message from XML to MRM:

```
<Name3><Name31>Value31</Name31>Value32</Name3>
```

produces a name element Name3, and a name-value element called Name31 with the value Value31. The second XML pcdta (Value32) cannot be represented and is discarded.

The unlike-parser-copy scans the source tree, and copies folders, also known as name elements, and leaf name-value pairs. Everything else, including elements flagged as *special* by the source parser, is not copied.

An example of a tree structure resulting in an unlike-parser-copy is:

```
SET OutputRoot.MRM.Data.Account = InputRoot.XML.Data.Account;
```

If the algorithm used to make an unlike-parser-copy does not suit your tree structure, you might need to further qualify the source field to restrict the amount of tree copied.

Be careful when you copy information from input messages to output messages in different domains. It is possible to code ESQL that creates a message structure or content that is not completely consistent with the rules of the parser that will process the output message. This can result in an output message not being

created, or being created with unexpected content. If you believe that the output message generated by a particular message flow does not contain the correct content, or have the expected form, check the ESQL that creates the output message, and look for potential mismatches of structure, field types, field names, and field values.

When copying trees between unlike parsers, you might also want to set the message format of the target parser. For example, if a message set has been defined with XML and CWF formats the following commands are required to copy an input XML stream to the MRM parser and set the latter to output in CWF format:

```
-- Copy message to the output, moving from XML to MRM domains
SET OutputRoot.MRM = InputRoot.XML;

-- Set the CWF format for output by the MRM domain
SET OutputRoot.Properties.MessageType = '<MessageTypeName>';
SET OutputRoot.Properties.MessageSet = '<MessageSetName>';
SET OutputRoot.Properties.MessageFormat = 'CWF';
```

Adding keywords to ESQL files

Keywords can be included in ESQL files in three ways:

comment fields

Add the keyword as a comment in the ESQL file:

```
-- $MQSI compiled by = John MQSI$
```

static strings

Include the keyword as part of a static string in the ESQL file:

```
Set target = '$MQSI_target = production only MQSI$'
```

variable string

Include the keyword value as a variable string in the ESQL file:

```
$MQSI_VERSION=$id$MQSI$
```

For this example, when the message flow source is extracted from the file repository, the repository's plug-in has been configured to substitute the identifier *\$id\$* with the actual version number. The identifier value that is required depends on the capability and configuration of the repository, and is not part of WebSphere Message Brokers.

Accessing databases from ESQL

ESQL has a number of statements and functions for accessing databases:

- The “CALL statement” on page 813 invokes a stored procedure.
- The “DELETE FROM statement” on page 857 removes rows from a database table.
- The “INSERT statement” on page 864 adds a row to a database table.
- The “PASSTHRU function” on page 980 can be used to make complex selections.
- The “PASSTHRU statement” on page 872 can be used to invoke administrative operations (for example, creating a table).
- The “SELECT function” on page 954 retrieves data from a table.
- The “UPDATE statement” on page 885 changes one or more values stored in zero or more rows.

You can access user databases from Compute, Database, and Filter nodes.

Note: There is no difference between the database access capabilities of these nodes; their names are partly historical and partly based on typical usage. You can use the data in the databases to update or create messages; or use the data in the message to update or create data in the databases.

Note that:

- Any node that uses any of the ESQL database statements or functions must have its Data Source property set with the name (that is, the ODBC DSN) of a database. The database must be accessible, operational, and allow the broker to connect to it.
- All databases accessed from the same node must have the same ODBC functionality as the database specified on the node's Data Source property. This requirement is always satisfied if the databases are of the same type (for example, DB2 or Oracle), at the same level (for example, release 8.1 CSD3), and on the same platform. Other database combinations may or may not have the same ODBC functionality. If a node tries to access a database that does not have the same ODBC functionality as the database specified on the node's Data Source property, the broker issues an error message.
- All tables referred to in a single SELECT FROM clause must be in the same database.

You must ensure that suitable ODBC data sources have been created on the system on which the broker is running. If you have used the `mqsisetdbparms` command to set a user ID and password for a particular database, the broker uses these values to connect to the database. If you have not set a user ID and password, the broker uses the default database user ID and password that you supplied on the `mqsicreatebroker` command (as modified by any subsequent `mqsichangebroker` commands).

On z/OS systems, use the JCL member BIPSDBP in the customization data set `<hlq>.SBIPPROC` to perform the **`mqsisetdbparms`** command.

You must also ensure that the database user IDs have sufficient privileges to perform the operations your flow requires. Otherwise errors will occur at runtime.

Select the Throw exception on database error property check box and the Treat warnings as errors property check box, and set the Transaction property to Automatic, to provide maximum flexibility. You can then use the COMMIT and ROLLBACK statements for transaction control, and create handlers for dealing with errors.

- "Referencing columns in a database"
- "Selecting data from database columns" on page 207
- "Accessing multiple database tables" on page 209
- "Changing database content" on page 210
- "Checking returns to SELECT" on page 211
- "Committing database updates" on page 212
- "Invoking stored procedures" on page 212

Referencing columns in a database:

While the standard SQL SELECT syntax is supported for queries to an external database, there are a number of points to be borne in mind. You must prefix the name of the table with the keyword Database to indicate that the SELECT is to be targeted at the external database, rather than at a repeating structure in the message.

The basic form of database SELECT is:

```
SELECT ...
  FROM Database.TABLE1
  WHERE ...
```

If necessary, you can specify a schema name:

```
SELECT ...
  FROM Database.SCHEMA.TABLE1
  WHERE ...
```

where SCHEMA is the name of the schema in which the table TABLE1 is defined. Include the schema if the user ID under which you are running does not match the schema. For example, if your userID is USER1, the expression Database.TABLE1 is equivalent to Database.USER1.TABLE1. However, if the schema associated with the table in the database is db2admin, you must specify Database.db2admin.TABLE1. If you do not include the schema, and this does not match your current user ID, the broker generates a runtime error when a message is processed by the message flow.

If, as in the two previous examples, a data source is not specified, TABLE1 must be a table in the default database specified by the node's data source property. To access data in a database other than the default specified on the node's data source property, you must specify the data source explicitly. For example:

```
SELECT ...
  FROM Database.DataSource.SCHEMA.TABLE1
  WHERE ...
```

Qualify references to column names with either the table name or the correlation name defined for the table by the FROM clause. So, where you could normally execute a query such as:

```
SELECT column1, column2 FROM table1
```

you must write one of the following two forms:

```
SELECT T.column1, T.column2 FROM Database.table1 AS T
```

```
SELECT table1.column1, table1.column2 FROM Database.table1
```

This is necessary in order to distinguish references to database columns from any references to fields in a message that might also appear in the SELECT:

```
SELECT T.column1, T.column2 FROM Database.table1
  AS T WHERE T.column3 = Body.Field2
```

You can use the AS clause to rename the columns returned. For example:

```
SELECT T.column1 AS price, T.column2 AS item
  FROM Database.table1 AS T WHERE...
```

The standard select all SQL option is supported in the SELECT clause. If you use this option, you must qualify the column names with either the table name or the correlation name defined for the table. For example:

```
SELECT T.* FROM Database.Table1 AS T
```

When you use ESQL procedure and function names within a database query, the positioning of these within the call affects how these names are processed. If it is determined that the procedure or function affects the results returned by the query, it is not processed as ESQL and is passed as part of the database call.

This applies when attempting to use a function or procedure name with the column identifiers within the SELECT statement.

For example, if you use a CAST statement on a column identifier specified in the Select clause, this is used during the database query to determine the data type of the data being returned for that column. An ESQL CAST is not performed to that ESQL data type, and the data returned is affected by the database interaction's interpretation of that data type.

If you use a function or procedure on a column identifier specified in the WHERE clause, this is passed directly to the database manager for processing.

The examples in the subsequent topics illustrate how the results sets of external database queries are represented in WebSphere Message Broker. The results of database queries are assigned to fields in a message using a Compute node.

A column function is a function that takes the values of a single column in all the selected rows of a table or message and returns a single scalar result.

Selecting data from database columns:

You can configure a Compute, Filter, or Database node to select data from database columns and include it in an output message. The following example assumes that you have a database table called USERTABLE with two char(6) data type columns (or equivalent), called Column1 and Column2. The table contains two rows:

	Column1	Column2
Row 1	value1	value2
Row 2	value3	value4

Configure the Compute, Filter, or Database node to identify the database in which you have defined the table. For example, if you're using the default database (specified on the "data source" property of the node), right-click the node, select **Open ESQL**, and code the following ESQL statements in the module for this node:

```
SET OutputRoot = InputRoot;  
DELETE FIELD OutputRoot.*[<];  
SET OutputRoot.XML.Test.Result[] =  
  (SELECT T.Column1, T.Column2 FROM Database.USERTABLE AS T);
```

This produces the following output message:

```

<Test>
  <Result>
    <Column1>value1</Column1>
    <Column2>value2</Column2>
  </Result>
  <Result>
    <Column1>value3</Column1>
    <Column2>value4</Column2>
  </Result>
</Test>

```

To trigger the SELECT, send a trigger message with an XML body that is of the following form:

```

<Test>
  <Result>
    <Column1></Column1>
    <Column2></Column2>
  </Result>
  <Result>
    <Column1></Column1>
    <Column2></Column2>
  </Result>
</Test>

```

The exact structure of the XML is not important, but the enclosing tag must be <Test> to match the reference in the ESQL. If it is not, the ESQL statements result in top-level enclosing tags being formed, which is not valid XML.

If you want to create an output message that includes all the columns of all the rows that meet a particular condition, use the SELECT statement with a WHERE clause:

```

-- Declare and initialize a variable to hold the
-- test value (in this case the surname Smith)
DECLARE CurrentCustomer STRING 'Smith';

-- Loop through table records to extract matching information
SET OutputRoot.XML.Invoice[] =
  (SELECT R FROM Database.USERTABLE AS R
   WHERE R.Customer.LastName = CurrentCustomer
  );

```

The message fields are created in the same order as the columns appear in the table.

If you are familiar with SQL in a database environment, you might expect to code SELECT *. This is not accepted by the broker because you must start all references to columns with a correlation name. This avoids ambiguities with declared variables. Also, if you code SELECT I.*, this is accepted by the broker but the * is interpreted as the first child element, not all elements, as you might expect from other database SQL.

Selecting data from a table in a case sensitive database system:

If the database system is case sensitive, you must use an alternative approach. This approach is also necessary if you want to change the name of the generated field to something different:


```

SET OutputRoot = InputRoot;
SET OutputRoot.XML.Test.Result[] =
  (SELECT T.Column1 AS Column1, T.Column2 AS Column2
   FROM Database.USERTABLE AS T);

```

This example produces the same message as the example above. Ensure that references to the database columns (in this example, T.Column1 and T.Column2) are specified in the correct case to match the database definitions exactly. If you do not do so, for example if you specify T.COLUMN1, the broker generates a runtime error. Note the use of Column1 and Column2 in the SELECT statement. You can use any values here, they do not have to match the names of the columns that you have defined in the database as they do in this example.

Accessing multiple database tables:

You can refer to multiple tables that you have created in the same database. Use the FROM clause on the SELECT statement to join the data from the two tables.

The following example assumes that you have two database tables called USERTABLE1 and USERTABLE2. Both tables have two char(6) data type columns (or equivalent).

USERTABLE1 contains two rows:

	Column1	Column2
Row 1	value1	value2
Row 2	value3	value4

USERTABLE2 contains two rows:

	Column3	Column4
Row 1	value5	value6
Row 2	value7	value8

All tables referenced by a single SELECT function must be in the same database. The database can be either the default (specified on the “data source” property of the node) or another database (specified on the FROM clause of the SELECT function).

Configure the Compute, Filter, or Database node that you’re using to identify the database in which you have defined the tables. For example, if you’re using the default database, right-click the node, select **Open ESQL**, and code the following ESQL statements in the module for this node:

```

SET OutputRoot.XML.Test.Result[] =
  (SELECT A.Column1 AS FirstColumn,
         A.Column2 AS SecondColumn,
         B.Column3 AS ThirdColumn,
         B.Column4 AS FourthColumn
   FROM Database.USERTABLE1 AS A,
         Database.USERTABLE2 AS B
   WHERE A.Column1 = 'value1' AND
         B.Column4 = 'value8'
  );

```

This results in the following output message content:

```
<Test>
  <Result>
    <FirstColumn>value1</FirstColumn>
    <SecondColumn>value2</SecondColumn>
    <ThirdColumn>value7</ThirdColumn>
    <FourthColumn>value8</FourthColumn>
  </Result>
</Test>
```

The example above shows how to access data from two database tables. You can code more complex FROM clauses to access multiple database tables (although all the tables must be in the same database). You can also refer to one or more message trees, and can use SELECT to join tables with tables, messages with messages, or tables with messages. “Joining data from XML messages and database tables” on page 262 provides an example of how to merge message data with data in a database table.

(defined by the *data source* property of the node).

If you specify an ESQL function or procedure on the column identifier in the WHERE clause, this is processed as part of the database query and not as ESQL.

Consider the following example:

```
SET OutputRoot.XML.Test.Result =
  THE(SELECT ITEM T.Column1 FROM Database.USERTABLE1 AS T
  WHERE UPPER(T.Column2) = 'VALUE2');
```

This attempts to return the rows where the value of Column2 converted to upper case is VALUE2. However, only the database manager can determine the value of T.Column2 for any given row, and therefore it cannot be processed by ESQL before the database query is issued, because the WHERE clause determines the rows that are returned to the message flow.

Therefore, the UPPER is passed to the database manager to be included as part of its processing. However, if the database manager cannot process the token within the select statement, an error is returned.

Changing database content:

You can code ESQL in the Compute, Database, and Filter nodes to change the contents of a database in the following ways:

- Update data in a database
- Insert data into a database
- Delete data from a database

The following ESQL code includes statements that show all three operations. This code is appropriate for a Database and Filter node; if you create this code for a Compute node, use the correlation name InputRoot in place of Root.

```

IF Root.XML.TestCase.Action = 'INSERT' THEN
  INSERT INTO Database.STOCK (STOCK_ID, STOCK_DESC, STOCK_QTY_HELD,
    BROKER_BUY_PRICE, BROKER_SELL_PRICE, STOCK_HIGH_PRICE, STOCK_HIGH_DATE,
    STOCK_HIGH_TIME) VALUES
    (CAST(Root.XML.TestCase.stock_id AS INTEGER),
    Root.XML.TestCase.stock_desc,
    CAST(Root.XML.TestCase.stock_qty_held AS DECIMAL),
    CAST(Root.XML.TestCase.broker_buy_price AS DECIMAL),
    CAST(Root.XML.TestCase.broker_sell_price AS DECIMAL),
    Root.XML.TestCase.stock_high_price,
    CURRENT_DATE,
    CURRENT_TIME);

ELSEIF Root.XML.TestCase.Action = 'DELETE' THEN

  DELETE FROM Database.STOCK WHERE STOCK.STOCK_ID =
    CAST(Root.XML.TestCase.stock_id AS INTEGER);

  ELSEIF Root.XML.TestCase.Action = 'UPDATE' THEN

    UPDATE Database.STOCK as A SET STOCK_DESC = Root.XML.TestCase.stock_desc
    WHERE A.STOCK_ID = CAST(Root.XML.TestCase.stock_id AS INTEGER);
END IF;

```

Checking returns to SELECT:

If a SELECT statement returns no data, or no further data, this is handled as a normal situation and no error code is set in SQLCODE. This occurs regardless of the setting of the *Throw Exception On Database Error* and *Treat Warnings As Errors* properties on the current node.

To recognize that a SELECT statement has returned no data, include ESQL that checks what has been returned. You can do this in a number of ways:

1. EXISTS

This returns a boolean value that indicates if a SELECT function returned one or more values (TRUE), or none (FALSE).

```

IF EXISTS(SELECT T.MYCOL FROM Database.MYTABLE) THEN
  ...

```

2. CARDINALITY

If you expect an array in response to a SELECT, you can use CARDINALITY to calculate how many entries have been received.

```

SET OutputRoot.XML.Testcase.Results[] = (
  SELECT T.MYCOL FROM Database.MYTABLE)
.....
IF CARDINALITY (OutputRoot.XML.Testcase.Results[]) > 0 THEN
  .....

```

3. IS NULL

If you have used either THE or ITEM keywords in your SELECT statement, a scalar value is returned. If no rows have been returned, the value set is NULL. However, it is possible that the value NULL is contained within the column, and you might want to distinguish between these two cases.

To do this include COALESCE in the SELECT statement, for example:

```

SET OutputRoot.XML.Testcase.Results VALUE = THE (
  SELECT ITEM COALESCE(T.MYCOL, 'WAS NULL')
  FROM Database.MYTABLE);

```

If this returns the character string WAS NULL, this indicates that the column contained NULL, and not that no rows were returned.

In previous releases, an SQLCODE of 100 was set in most cases if no data, or no further data, was returned. An exception was raised by the broker if you chose to handle database errors in the message flow.

Committing database updates:

When you create a message flow that interacts with databases, you can choose whether the updates that you make are committed when the current node has completed processing, or when the current invocation of the message flow has terminated.

For each node, select the appropriate option for the *Transaction* property to specify when its database updates are to be committed:

- Choose *Automatic* (the default) if you want updates made in this node to be committed or rolled back as part of the whole message flow. The actions that you define in the ESQL module are performed on the message and it continues through the message flow. If the message flow completes successfully, the updates are committed. If the message flow fails, the message and the database updates are rolled back.
- Choose *Commit* if you want to commit the action of the node on the database, irrespective of the success or failure of the message flow as a whole. The database update is committed when the node processing is successfully completed, that is, after all ESQL has been processed, even if the message flow itself detects an error in a subsequent node that causes the message to be rolled back.

The value that you choose is implemented for the database tables that you have updated. You cannot select a different value for each table.

If you have set *Transaction* to *Commit*, the behavior of the message flow and the commitment of database updates can be affected by the use of the PROPAGATE statement.

If you choose to include a PROPAGATE statement in the node's ESQL that generates one or more output message from the node, the processing of the PROPAGATE statement is not considered complete until the entire path that the output message takes has completed. This path might include several other nodes, including one or more output nodes. Only then does the node that issues the PROPAGATE statement receive control back and its ESQL terminate. At that point, a database commit is performed, if appropriate.

If one of the nodes on the propagated path detects an error and throws an exception, the processing of the node in which you have coded the PROPAGATE statement never completes. If the error processing results in a rollback, the message flow and the database update in this node are rolled back. This behavior is consistent with the stated operation of the *Commit* option, but might not be the behavior that you expect.

Invoking stored procedures:

To invoke a procedure that is stored in a database, use the ESQL CALL statement. The stored procedure must be defined by a "CREATE PROCEDURE statement" on page 837 that has:

- A Language clause of DATABASE
- An EXTERNAL NAME clause that identifies the name of the procedure in the database and, optionally, the database schema to which it belongs.

When you invoke a stored procedure with the CALL statement, the broker ensures that the ESQL definition and the database definition match:

- The external name of the procedure must match a procedure in the database.
- The number of parameters must be the same.
- The type of each parameter must be the same.
- The direction of each parameter (IN, OUT, INOUT) must be the same.

The following restrictions apply to the use of stored procedures:

- Overloaded procedures are not supported. (An overloaded procedure is one that has the same name as another procedure in the same database schema with a different number of parameters, or parameters with different types.) If the broker detects that a procedure has been overloaded, it raises an exception.
- In an Oracle stored procedure declaration, you are not permitted to constrain CHAR and VARCHAR2 parameters with a length, and NUMBER parameters with a precision or scale, or both. Use %TYPE when you declare CHAR, VARCHAR and NUMBER parameters to provide constraints on a formal parameter.

!
!
!
!
!

Creating a stored procedure in ESQL:

When you define an ESQL procedure that corresponds to a database stored procedure, you can specify either a qualified name (where the qualifier is a database schema) or an unqualified name.

To create a stored procedure:

1. Code a statement similar to this example to create an unqualified procedure:

```
CREATE PROCEDURE myProc1(IN p1 CHAR) LANGUAGE DATABASE EXTERNAL NAME "myProc";
```

The EXTERNAL NAME that you specify must match the definition you have created in the database, but you can specify any name you choose for the corresponding ESQL procedure.
2. Code a statement similar to this example to create a qualified procedure:

```
CREATE PROCEDURE myProc2(IN p1 CHAR) LANGUAGE DATABASE EXTERNAL NAME "Schema1.myProc";
```
3. Code a statement similar to this example to create a qualified procedure in an Oracle package:

```
CREATE PROCEDURE myProc3(IN p1 CHAR) LANGUAGE DATABASE EXTERNAL  
NAME "mySchema.myPackage.myProc";
```

For examples of stored procedure definitions in the database, see the “CREATE PROCEDURE statement” on page 837.

Calling a stored procedure:

1. Code a statement similar to this example to invoke an unqualified procedure:

```
CALL myProc1('HelloWorld');
```

Because it is not defined explicitly as belonging to any schema, the myProc1 procedure must exist in the default schema (the name of which is the user name used to connect to the data source) or the command fails.
2. The following example calls the myProc procedure in schema Schema1.

```
CALL myProc2('HelloWorld');
```
3. Code a statement similar to this example to invoke an unqualified procedure with a dynamic schema:

```
DECLARE Schema2 char 'mySchema2';  
CALL myProc1('HelloWorld') IN Database.{Schema2};
```

This statement calls the myProc1 procedure in database Schema2, overriding the default "username" schema.

Calling a stored procedure that returns two result sets:

To call a stored procedure that takes one input parameter and returns one output parameter and two result sets:

1. Define the procedure with a CREATE PROCEDURE statement that specifies one input parameter, one output parameter, and two result sets:

```
CREATE PROCEDURE myProc1 (IN P1 INT, OUT P2 INT)
LANGUAGE DATABASE
DYNAMIC RESULT SETS 2
EXTERNAL NAME "myschema.myproc1";
```

2. To invoke the myProc1 procedure using a field reference, code:

```
/* using a field reference */
CALL myProc1(InVar1, OutVar2, Environment.ResultSet1[],
            OutputRoot.XML.Test.ResultSet2[]);
```

3. To invoke the myProc1 procedure using a reference variable, code:

```
/* using a reference variable*/
DECLARE cursor REFERENCE TO OutputRoot.XML.Test;

CALL myProc1(InVar1, cursor.OutVar2, cursor.ResultSet1[],
            cursor.ResultSet2[]);
```

Coding ESQL to handle errors

Introduction

When processing messages in message flows, errors can be due to:

1. External causes. For example, the incoming message is syntactically invalid, a database used by the flow has been shut down, or the power supply to the machine on which the broker is running fails.
2. Internal causes. For example, an attempt to insert a row into a database table fails because of a constraint check, or a character string read from a database cannot be converted to a number because it contains alphabetic characters.
Internal errors can be caused by programs storing invalid data in the database or by a flaw in the logic of a flow.

The message flow designer must give errors serious consideration and decide how they are to be handled.

Using default error-handling

The simplest strategy for handling ESQL errors is to do nothing and use the broker's default behavior. The default behavior is to cut short the processing of the failing message and to proceed to the next message. Input and output nodes provide options to control exactly what happens when processing is cut short.

If the input and output nodes are set to transactional mode, the broker restores the state prior to the message being processed:

1. The input message that has apparently been taken from the input queue is put back.
2. Any output messages that the flow has apparently written to output queues are discarded.

If the input and output nodes are not set to transactional mode:

1. The input message taken from the input queue is not put back.

2. Any output messages that the flow has written to output queues remain there.

Each of these strategies has its advantages. The transactional model preserves the consistency of data, while the non-transactional model maximizes the continuity of message processing. Remember that in the transactional model the failing input message is put back on to the input queue and the broker will attempt to process it again. The most likely outcome of this is that the message continues to fail until the retry limit is reached, at which point it is placed on a dead letter queue. The reason for the failure to process the message is logged to the system event log (Windows) or syslog (UNIX). Thus the failing message holds up the processing of subsequent good messages for a while and is then left unprocessed by the broker.

Most databases operate transactionally, so that all changes made to database tables are committed if the processing of the message succeeds and rolled back if it fails, thus maintaining the integrity of data. An exception to this is if the broker itself, or a database, fails. (For example, the power to the machines they are running on could be interrupted.) It is possible in these cases for changes in some databases to be committed and changes in others not; or for the database changes to be committed but the input and output messages not to be committed. If these possibilities concern you, the flow should be made coordinated and the databases involved configured for this way of working.

Using customized error handling: Here are some general tips about creating customized error handlers:

1. If you require something better than default error handling, the first step is to use a handler (see “DECLARE HANDLER statement” on page 856). Create one handler per node, to intercept all possible exceptions (or as many exceptions as can be foreseen).
2. Having intercepted an error, the error handler can use whatever logic is appropriate to handle it. Alternatively, it can use a THROW statement or node to create an exception, which could be handled higher in the flow logic or even reach the input node, causing the transaction to be rolled back. See “Throwing an exception” on page 218.
3. If a node throws an exception that is not caught by the handler, the flow is diverted to the failure terminal, if one is attached, or handled by default error-handling if not.

It is recommended that you use failure terminals to catch unhandled errors. Attach a simple logic flow to the failure terminal. This logic flow could consist of a database or compute node that writes a log record to a database (possibly including the message’s bit-stream) or writes a record to the event log. It could also contain an output node that writes the message to a special queue.

The full exception tree is passed to any node connected to a failure terminal. (The exception tree is described in “ExceptionList tree” on page 21.)

4. Your error handlers are responsible for logging each error in an appropriate place, such as the system event log.

For a detailed discussion of the options that you can use to process errors in a message flow, see “Handling errors in message flows” on page 111. The following topics provide examples of what you can do:

- “Throwing an exception” on page 218
- “Capturing database state” on page 219

Coding to detect errors

The following sections assume that it is the broker that detects the error. It is quite possible, however, for the logic of the flow to detect an error. For example, when coding the flow logic you could use:

- IF statements inserted specifically to detect situations that should not occur
- The ELSE clause of a case expression or statement to trap routes through the code that should not be possible

As an example of a flow logic-detected error, consider a field that has a range of possible integer values indicating the type of message. In such a case it would not be good practice to leave to chance what would happen if a message were to arrive in which the field's value did not correspond to any known type of message. One way this could happen is if the system is upgraded to support extra types of message but one part of the system is not upgraded.

Using your own logic to handle input messages that are not valid

Input messages that are syntactically invalid (and input messages that appear to be not valid because of erroneous message format information) are difficult to deal with because the broker has no idea what the message contains. Probably the best way of dealing with them is to configure the input node to fully parse and validate the message.

Note, however, that this applies only to predefined messages, that is, MRM or IDOC.

If the input node is configured in this way, the following is guaranteed if the input message cannot be parsed successfully:

- The input message never emerges from the node's normal output terminal (it goes to the failure terminal).
- The input message never enters the main part of the message flow.
- The input message never causes any database updates.
- No messages are written to any output queues.

To deal with a failing message, connect a simple logic flow to the failure terminal.

The only disadvantage to this strategy is that, if the normal flow does not require access to all the message's fields, the forcing of complete parsing of the message affects performance.

Using your own logic to handle database errors

Database errors fall into three categories:

1. The database isn't working at all (for example, it's off line).
2. The database is working but refuses your request (for example, a lock contention occurs).
3. The database is working but what you ask it to do is impossible (for example, to read from a non-existent table).

If you require something better than default error handling, the first step is to use a handler (see "DECLARE HANDLER statement" on page 856) to intercept the exception. The handler can determine the nature of the failure from the SQL state returned by the database.

Database not working

If a database isn't working at all and is essential to the processing of messages, there is probably not much you can do. In this case the handler, having determined the cause, might do any of the following:

- Use the RESIGNAL statement to re-throw the original error, thus allowing the default error handler to take over.
- Use a different database.
- Write the message to a special output queue.

However, take care with this sort of strategy. Because the handler absorbs the exception, any changes to other databases or writes to queues are committed.

Database refuses your request

The situation when a lock contention occurs is similar to the "Database not working" case. This is because the database will have backed out *all* the database changes you have made for the current message, not just the failing request. Therefore, unless you are sure this was the only update, it is unlikely that there is any better strategy than default error-handling, except possibly logging the error or passing the message to a special queue.

Impossible requests

The case where the database is working but what you ask it to do is impossible covers a wide variety of problems.

If, as in the example, the database simply doesn't have a table of the name the flow expects, it is again unlikely that there is any better strategy than default error-handling, except possibly logging the error or passing the message to a special queue.

Many other errors may be handled successfully, however. For example, an attempt to insert a row might fail because there is already such a row and the new row would be a duplicate. Or an attempt to update a row might fail because there is no such row (that is, the update updated zero rows). In these cases, the handler can incorporate whatever logic you think fit. It might insert the missing row or utilize the existing one (possibly making sure the values in it are suitable).

Note: For an update of zero rows to be reported as an error the node property "treat warnings as errors" must be set to true. This is not the default setting.

Using your own logic to handle errors in output nodes

Errors occurring in MQ output nodes report the nature of the error in the SQL state and give additional information in the *SQL native error* variable. Thus, if something better than default error handling is required, the first step is to use a handler (see "DECLARE HANDLER statement" on page 856) to intercept the exception. Such a handler might well surround only a single PROPAGATE statement.

Using your own logic to handle other errors

Besides those covered above, a variety of other errors can occur. For example, an arithmetic calculation might overflow, a cast might fail because of the unsuitability of the data, or an access to a message field might fail because of a type constraint. The broker offers two programming strategies for dealing with these.

1. The error causes an exception that is either handled or left to roll back the transaction.
2. The failure is recorded as a special value that is tested for later.

In the absence of a type constraint, an attempt to access a non-existent message field results in the value null. Null values propagate through expressions, making the result null. Thus, if an expression, however complex, does not return a null value, you know that all the values it needed to calculate its result were not null.

Cast expressions can have a default clause. If there is a default clause, casts fail quietly; instead of throwing an exception, they simply return the default value. The default value could be an innocuous number (for example, zero for an integer), or a value that is clearly invalid in the context (for example, -1 for a customer number). Null might be particularly suitable, because it is a value that is different from all others and that will propagate through expressions without any possibility of the error condition being masked.

Handling errors in other nodes

Exceptions occurring in other nodes (that is, downstream of a PROPAGATE statement) might be caught by handlers in the normal way. Handling such errors intelligently, however, poses the special problem that, as another node was involved in the original error, another node, and not necessarily the originator of the exception, is very likely to be involved in handling it.

To help in these situations the database and compute nodes have four new terminals called out1, out2, out3, and out4. In addition the syntax of the “PROPAGATE statement” on page 874 has been extended to include target expression, message source and control clauses to give more control over these extra terminals.

Throwing an exception:

If you detect an error or other situation in your message flow in which you want message processing to be terminated, you can throw an exception in a message flow in two ways:

1. Use the ESQL THROW EXCEPTION statement.
Include the THROW statement anywhere in the ESQL module for a Compute, Database, or Filter node. Use the options on the statement to code your own data to be inserted into the exception.
2. Include a THROW node in your message flow.
Set the node properties to identify the source and content of the exception.

Using either statement options or node properties, you can specify a message identifier and values that are inserted into the message text to give additional information and identification to users who interpret the exception. You can specify any message in any catalog that is available to the broker. See Using event logging from a user-defined extension for more information.

The situations in which you might want to throw an exception are determined by the behavior of the message flow; decide when you design the message flow where this action might be appropriate. For example, you might want to examine the content of the input message to ensure that it meets criteria that cannot be detected by the input node (which might check that a particular message format is received).

The example below uses the example Invoice message to show how you can use the ESQL THROW statement. If you want to check that the invoice number is within a particular range, throw an exception for any invoice message received that does not fall in the valid range.

```
--Check for invoice number lower than permitted range
IF Body.Invoice.InvoiceNo < 100000 THEN
    THROW USER EXCEPTION CATALOG 'MyCatalog' MESSAGE 1234 VALUES
    ('Invoice number too low', Body.Invoice.InvoiceNo);

-- Check for invoice number higher than permitted range
ELSEIF Body.InvoiceNo > 500000 THEN
    THROW USER EXCEPTION CATALOG 'MyCatalog' MESSAGE 1235 VALUES
    ('Invoice number too high', Body.Invoice.InvoiceNo);

ELSE DO
    -- invoice number is within permitted range
    -- complete normal processing
ENDIF;
```

Capturing database state:

If an error occurs when accessing an external database, you have two options:

- Let the broker throw an exception during node processing
- Process the exception within the node itself using ESQL statements

The first option is the default; ESQL processing in the current node is abandoned. The exception is then propagated backwards through the message flow until an enclosing catch node, or the input node for this message flow, is reached. If the exception reaches the input node, any transaction is rolled back.

The second option requires an understanding of database return codes and a logical course of action to take when an error occurs. To enable this inline database error processing, you must clear the Filter, Database, or Compute node's *Throw Exception On Database Error* property. If you do this, the node sets the database state indicators SQLCODE, SQLSTATE, SQLNATIVEERROR, and SQLERRORTXT, with appropriate information from the database manager instead of throwing an exception.

The indicators contain information only when an error (not a warning) occurs, unless you have selected the *Treat Warnings As Errors* property. In the case of successful and success with information database operations, the indicators contain their default success values.

You can use the values contained in these indicators in ESQL statements to make decisions about the action to take. You can access these indicators with the SQLCODE, SQLSTATE, SQLNATIVEERROR, and SQLERRORTXT functions.

If you are attempting inline error processing, you must check the state indicators after each database statement is executed to ensure that you catch and assess all errors. When processing the indicators, if you meet an error that you cannot handle inline, you can raise a new exception either to deal with it upstream in a catch node, or to let it through to the input node so that the transaction is rolled back. You can use the ESQL THROW statement to do this.

You might want to check for the special case in which a SELECT returns no data. This situation is not considered an error and SQLCODE is not set, so you must test explicitly for it. This is described in "Checking returns to SELECT" on page 211.

Using ESQL to access database state indicators

The following ESQL example shows how to use the four database state functions, and how to include the error information that is returned in an exception:

```
DECLARE SQLState1 CHARACTER;
DECLARE SQLErrorText1 CHARACTER;
DECLARE SQLCode1 INTEGER;
DECLARE SQLNativeError1 INTEGER;

-- Make a database insert to a table that does not exist --
INSERT INTO Database.DB2ADMIN.NONEXISTENTTABLE (KEY,QMGR,QNAME)
        VALUES (45,'REG356','my TESTING 2');

--Retrieve the database return codes --
SET SQLState1 = SQLSTATE;
SET SQLCode1 = SQLCODE;
SET SQLErrorText1 = SQLERRORTXT;
SET SQLNativeError1 = SQLNATIVEERROR;

--Use the THROW statement to back out the database and issue a user exception--
THROW USER EXCEPTION MESSAGE 2950 VALUES
( 'The SQL State' , SQLState1 , SQLCode1 , SQLNativeError1 ,
SQLErrorText1 );
```

You do not have to throw an exception when you detect a database error; you might prefer to save the error information returned in the LocalEnvironment tree, and include a Filter node in your message flow that routes the message to error or success subflows according to the values saved.

The Airline Reservations sample program provides another example of ESQL that uses these database functions.

Manipulating messages in the MRM domain

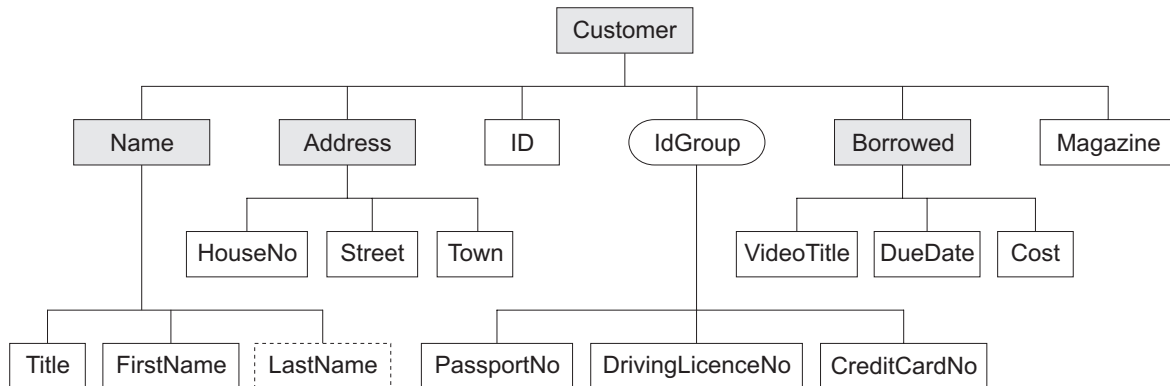
The following topics tell you how to deal with messages that have been modeled in the MRM domain, and that are parsed by the MRM parser. The physical formats associated with the message models do not affect this information unless specifically stated. Use this information in conjunction with the information in the topics in “Manipulating message body content” on page 171.

- “Accessing elements in a message in the MRM domain” on page 221
- “Accessing multiple occurrences of an element in a message in the MRM domain” on page 222
- “Accessing attributes in a message in the MRM domain” on page 223
- “Accessing elements within groups in a message in the MRM domain” on page 224
- “Accessing mixed content in a message in the MRM domain” on page 226
- “Accessing embedded messages in the MRM domain” on page 227
- “Accessing the content of a message in the MRM domain with namespace support enabled” on page 228
- “Querying null values in a message in the MRM domain” on page 229
- “Setting null values in a message in the MRM domain” on page 229
- “Working with MRM messages and bit streams” on page 233
- “Handling large MRM messages” on page 235

If you have migrated message sets from WebSphere MQ Integrator Broker Version 2.1 or WebSphere MQ Event Broker Version 2.1, you might also need to refer to the information in the following section:

- “Accessing objects in migrated message models” on page 230

The following diagram shows the structure of the message, Customer, that is used in the Video Rental sample; it is used in the samples in the topics listed above to show ESQL that manipulates the objects that can be defined in a message model.



The message includes a variety of structures that demonstrate the ways in which metadata can be classified to the MRM. Within an MRM message set, you can define the following objects: messages, types, groups, elements, and attributes. Folder icons that represent each of these types of objects are displayed for each message definition file in the Broker Application Development perspective.

Each message definition file can contribute to a namespace; in this sample, each namespace is completely defined by a single message definition file. You can combine several message definition files to form a complete message dictionary, which you can then deploy to a broker.

The video sample has three message definition files:

Customer.mxsd

Resides in the noTarget namespace

Address.mxsd

Resides in the namespace <http://www.ibm.com/AddressDetails>

Borrowed.mxsd

Resides in the namespace <http://www.ibm.com/BorrowedDetails>

Refer to the Video Rental message structure for detailed information about the objects that are defined in this message model.

Accessing elements in a message in the MRM domain:

You can use ESQL to manipulate the logical tree that represents a message in the message flow. This topic describes how to access data for elements in a message in the MRM domain.

You can populate an element with data with the SET statement:

```
SET OutputRoot.MRM.Name = UPPER(InputRoot.MRM.Name);
```

The field reference on the left hand side of the expression refers to the element called Name within the MRM message domain. This statement takes the input value for the Name field, converts it to uppercase, and assigns the result to the same element in the output message.

The Name element is defined in the noTarget namespace. No namespace prefix is specified in front of the Name part of the field reference in the example above. If you have defined an MRM element in a namespace other than the noTarget namespace, you must also specify a namespace prefix in the statement. For example:

```
DECLARE brw NAMESPACE 'http://www.ibm.com/Borrowed';  
  
SET OutputRoot.MRM.brw:Borrowed.VideoTitle = 'MRM Greatest Hits';
```

For more information about using namespaces with messages in the MRM domain, see “Accessing the content of a message in the MRM domain with namespace support enabled” on page 228.

Accessing multiple occurrences of an element in a message in the MRM domain:

You can access MRM domain elements following the general guidance given in “Accessing known multiple occurrences of an element” on page 176 and “Accessing unknown multiple occurrences of an element” on page 177. Further information specific to MRM domain messages is provided in this topic.

Consider the following statements:

```
DECLARE brw NAMESPACE 'http://www.ibm.com/Borrowed';  
  
SET OutputRoot.MRM.brw:Borrowed[1].VideoTitle = 'MRM Greatest Hits Volume 1';  
SET OutputRoot.MRM.brw:Borrowed[2].VideoTitle = 'MRM Greatest Hits Volume 2';
```

The above SET statements operate on two occurrences of the element Borrowed. Each statement sets the value of the child VideoTitle. The array index indicates which occurrence of the repeating element you are interested in.

When you define child elements of a complex type (which has its *Composition* property set to Sequence) in a message set, you can add the same element to the complex type more than once. These instances do not have to be contiguous, but you must use the same method (array notation) to refer to them in ESQL.

For example, if you create a complex type with a *Composition* of Sequence that contains the following elements:

```
StringElement1  
IntegerElement1  
StringElement1
```

use the following ESQL to set the value of StringElement1:

```
SET OutputRoot.MRM.StringElement1[1] =  
    'This is the first occurrence of StringElement1';  
SET OutputRoot.MRM.StringElement1[2] =  
    'This is the second occurrence of StringElement1';
```

You can also use the arrow notation (the greater than (>) and less than (<) symbols) to indicate the direction of search and the index to be specified:

```

SET OutputRoot.MRM.StringElement1[>] =
    'This is the first occurrence of StringElement1';
SET OutputRoot.MRM.StringElement1[<2] =
    'This is the last but one occurrence of
    StringElement1';
SET OutputRoot.MRM.StringElement1[<1] =
    'This is the last occurrence of StringElement1';

```

Refer to “Accessing known multiple occurrences of an element” on page 176 and “Accessing unknown multiple occurrences of an element” on page 177 for additional detail.

Accessing attributes in a message in the MRM domain:

When an MRM message is parsed into a logical tree, attributes and the data that they contain are created as name-value pairs in the same way that MRM elements are. This means that the ESQL that you code to interrogate and update the data held in attributes refers to the attributes in a similar manner.

Consider the Video sample MRM message. The attribute LastName is defined as a child of the Name element in the Customer message. Here is an example input XML message:

```

<Customer xmlns:addr="http://www.ibm.com/AddressDetails"
xmlns:brw="http://www.ibm.com/BorrowedDetails">
  <Name LastName="Bloggs">
    <Title>Mr</Title>
    <FirstName>Fred</FirstName>
  </Name>
  <addr:Address>
    <HouseNo>13</HouseNo>
    <Street>Oak Street</Street>
    <Town>Southampton</Town>
  </addr:Address>
  <ID>P</ID>
  <PassportNo>J123456TT</PassportNo>
  <brw:Borrowed>
    <VideoTitle>Fast Cars</VideoTitle>
    <DueDate>2003-05-23T01:00:00</DueDate>
    <Cost>3.50</Cost>
  </brw:Borrowed>
  <brw:Borrowed>
    <VideoTitle>Cut To The Chase</VideoTitle>
    <DueDate>2003-05-23T01:00:00</DueDate>
    <Cost>3.00</Cost>
  </brw:Borrowed>
  <Magazine>0</Magazine>
</Customer>

```

When the input message is parsed, values are stored in the logical tree as shown in the following section of user trace:

```

(0x0100001B):MRM = (
  (0x01000013):Name = (
    (0x0300000B):LastName = 'Bloggs'
    (0x0300000B):Title = 'Mr'
    (0x0300000B):FirstName = 'Fred'
  )
  (0x01000013)http://www.ibm.com/AddressDetails:Address = (
    (0x0300000B):HouseNo = 13
    (0x0300000B):Street = 'Oak Street'
    (0x0300000B):Town = 'Southampton'
  )
  (0x0300000B):ID = 'P'
  (0x0300000B):PassportNo = 'J123456TT'
  (0x01000013)http://www.ibm.com/BorrowedDetails:Borrowed = (
    (0x0300000B):VideoTitle = 'Fast Cars'
    (0x0300000B):DueDate = TIMESTAMP '2003-05-23 00:00:00'
    (0x0300000B):Cost = 3.50
  )
  (0x01000013)http://www.ibm.com/BorrowedDetails:Borrowed = (
    (0x0300000B):VideoTitle = 'Cut To The Chase '
    (0x0300000B):DueDate = TIMESTAMP '2003-05-23 00:00:00'
    (0x0300000B):Cost = 3.00
  )
  (0x0300000B):Magazine = FALSE

```

The following ESQL changes the value of the LastName attribute in the output message:

```
SET OutputRoot.MRM.Name.LastName = 'Smith';
```

Be aware of the ordering of attributes when you code ESQL. When attributes are parsed, the logical tree inserts the corresponding name-value before the MRM element's child elements. In the previous example, the child elements Title and FirstName appear in the logical message tree after the attribute LastName. In the Broker Application Development perspective, the Outline view displays attributes after the elements. When you code ESQL to construct output messages, you must define name-value pairs for attributes before any child elements.

Accessing elements within groups in a message in the MRM domain:

When an input message is parsed, structures that you have defined as groups in your message set are not represented in the logical tree, but its children are. If you want to refer to or update values for elements that are children of a groups, do not include the group in the ESQL statement. Groups do not have tags that appear in instance messages, and do not appear in user trace of the logical message tree.

Consider the following Video message:


```

<Customer xmlns:addr="http://www.ibm.com/AddressDetails"
xmlns:brw="http://www.ibm.com/BorrowedDetails">
  <Name LastName="Bloggs">
    <Title>Mr</Title>
    <FirstName>Fred</FirstName>
  </Name>
  <addr:Address>
    <HouseNo>13</HouseNo>
    <Street>Oak Street</Street>
    <Town>Southampton</Town>
  </addr:Address>
  <ID>P</ID>
  <PassportNo>J123456TT</PassportNo>
  <brw:Borrowed>
    <VideoTitle>Fast Cars</VideoTitle>
    <DueDate>2003-05-23T01:00:00</DueDate>
    <Cost>3.50</Cost>
  </brw:Borrowed>
  <brw:Borrowed>
    <VideoTitle>Cut To The Chase</VideoTitle>
    <DueDate>2003-05-23T01:00:00</DueDate>
    <Cost>3.00</Cost>
  </brw:Borrowed>
  <Magazine>0</Magazine>
</Customer>

```

When the input message is parsed, values are stored in the logical tree as shown in the following section of user trace:

```

(0x0100001B):MRM = (
  (0x01000013):Name = (
    (0x0300000B):LastName = 'Bloggs'
    (0x0300000B):Title = 'Mr'
    (0x0300000B):FirstName = 'Fred'
  )
  (0x01000013)http://www.ibm.com/AddressDetails:Address = (
    (0x0300000B):HouseNo = 13
    (0x0300000B):Street = 'Oak Street'
    (0x0300000B):Town = 'Southampton'
  )
  (0x0300000B):ID = 'P'
  (0x0300000B):PassportNo = 'J123456TT'
  (0x01000013)http://www.ibm.com/BorrowedDetails:Borrowed = (
    (0x0300000B):VideoTitle = 'Fast Cars'
    (0x0300000B):DueDate = TIMESTAMP '2003-05-23 00:00:00'
    (0x0300000B):Cost = 3.50
  )
  (0x01000013)http://www.ibm.com/BorrowedDetails:Borrowed = (
    (0x0300000B):VideoTitle = 'Cut To The Chase '
    (0x0300000B):DueDate = TIMESTAMP '2003-05-23 00:00:00'
    (0x0300000B):Cost = 3.00
  )
  (0x0300000B):Magazine = FALSE
)

```

Immediately following the element named ID, the MRM message definition uses a group which has a *Composition* of Choice. The group is defined with three children: PassportNo, DrivingLicenceNo, and CreditCardNo. The choice composition dictates that instance documents must use only one of these three possible alternatives. The example shown above uses the PassportNo element.

When you refer to this element in ESQL statements, you do not specify the group to which the element belongs. For example:

```
SET OutputRoot.MRM.PassportNo = 'J999999TT';
```

If you define messages within message sets that include XML and TDS physical formats, you can determine from the message data which option of a choice has been taken, because the tags in the message represent one of the choice's options. However, if your messages have CWF physical format, or are non-tagged TDS messages, it is not clear from the message data, and the application programs processing the message must determine which option of the choice has been selected. This is known as unresolved choice handling. For further information, see the description of the value of *Choice* in Complex type logical properties.

Accessing mixed content in a message in the MRM domain:

When you define a complex type in a message model, you can optionally specify its content to be mixed. This setting, in support of mixed content in XML schema, allows you to manipulate data that is included between elements in the message.

Consider the following example:

```
<MRM>
  <Mess1>
    abc
    <Elem1>def</Elem1>
    ghi
    <Elem2>jkl</Elem2>
    mno
    <Elem3>pqr</Elem3>
  </Mess1>
</MRM>
```

The strings *abc*, *ghi*, and *mno* do not represent the value of a particular element (unlike *def*, for example, which is the value of element *Elem1*). The presence of these strings means that you must model *Mess1* with mixed content. You can model this XML message in the MRM using the following objects:

Message

The message *Name* property is set to *Mess1* to match the XML tag.

The *Type* property is set to *tMess1*.

Type The complex type *Name* property is set to *tMess1*.

The *Composition* property is set to *OrderedSet*.

The complex type has mixed content.

The complex type contains the following objects:

Element

The *Name* property is set to *Elem1* to match the XML tag.

The *Type* property is set to simple type *xsd:string*.

Element

The *Name* property is set to *Elem2* to match the XML tag.

The *Type* property is set to simple type *xsd:string*.

Element

The *Name* property is set to *Elem3* to match the XML tag.

The *Type* property is set to simple type *xsd:string*.

If you code the following ESQL:

```
SET OutputRoot.MRM.*[1] = InputBody.Elem3;
SET OutputRoot.MRM.Elem1 = InputBody.*[5];
SET OutputRoot.MRM.*[3] = InputBody.Elem2;
SET OutputRoot.MRM.Elem2 = InputBody.*[3];
SET OutputRoot.MRM.*[5] = InputBody.Elem1;
SET OutputRoot.MRM.Elem3 = InputBody*[1];
```

the mixed content is successfully mapped to the following output message:

```
<MRM>
  <Mess1>
    pqr
    <Elem1>mno</Elem1>
    jk1
    <Elem2>ghi</Elem2>
    def
    <Elem3>abc</Elem3>
  </Mess1>
</MRM>
```

Accessing embedded messages in the MRM domain:

If you have defined a multipart message, you have at least one message embedded within another. Within the overall complex type that represents the outer messages, you can model the inner message in one of the following ways:

- An element (named *E_outer1* in the following example) with its *Type* property set to a complex type that has been defined with its *Composition* property set to *Message*
- A complex type with its *Composition* property set to *Message* (named *t_Embedded* in the following example)

The ESQL that you need to write to manipulate the inner message varies depending on which of the above models you have used. For example, assume that you have defined:

- An outer message *M_outer* that has its *Type* property set to *t_Outer*.
- An inner message *M_inner1* that has its *Type* set to *t_Inner1*
- An inner message *M_inner2* that has its *Type* set to *t_Inner2*
- Type *t_Outer* that has its first child element named *E_outer1* and its second child defined as a complex type named *t_Embedded*
- Type *t_Embedded* that has its *Composition* property set to *Message*
- Type *t_Inner1* that has its first child element named *E_inner11*
- Type *t_Inner2* that has its first child element named *E_inner21*
- Type *t_outer1* that has its *Composition* property set to *Message*
- Element *E_outer1* that has its *Type* property set to *t_outer1*

If you want to set the value of *E_inner11*, code the following ESQL:

```
SET OutputRoot.MRM.E_outer1.M_inner1.E_inner11 = 'FRED';
```

If you want to set the value of *E_inner21*, code the following ESQL:

```
SET OutputRoot.MRM.M_inner2.E_inner21 = 'FRED';
```

If you copy message headers from the input message to the output message, and your input message type contains a path, only the outermost name in the path is copied to the output message type.

When you configure a message flow to handle embedded messages, you can specify the path of a message type in either an MQRFH2 header (if one is present in the input message) or in the input node *Message Type* property in place of a name (for example, for the message modeled above, the path could be specified as M_Outer/M_Inner1/M_Inner2 instead of just M_Outer).

If you have specified that the input message has a physical format of either CWF or XML, any message type prefix is concatenated in front of the message type from the MQRFH2 or input node, giving a final path to use (for more information refer to Multipart messages). The MRM uses the first item in the path as the outermost message type, then progressively works inwards when it finds a complex type with its *Composition* property set to Message.

If you have specified that the input message has a physical format of TDS, a different process that uses message keys is implemented. This is described in TDS format: Multipart messages.

For more information about path concatenations, see Message set properties.

Accessing the content of a message in the MRM domain with namespace support enabled:

You can exploit namespace support for messages that are parsed by the MRM parser.

When you want to access elements of a message and namespaces are enabled, you must include the namespace when you code the ESQL reference to the element. If you do not do so, the broker searches the notarget namespace. If the element is not found in the notarget namespace, the broker searches all other known namespaces in the message dictionary (that is, within the deployed message set). For performance and integrity reasons, specify namespaces wherever they apply.

The most efficient way to refer to elements when namespaces are enabled is to define a namespace constant, and use this in the appropriate ESQL statements. This makes your ESQL code much easier to read and maintain.

Define a constant using the DECLARE NAMESPACE statement:

```
DECLARE ns01 NAMESPACE 'http://www.ns01.com'  
.  
.  
SET OutputRoot.MRM.Element1 = InputBody.ns01:Element1;
```

ns01 is interpreted correctly as a namespace because of the way that it is declared.

You can also use a CHARACTER variable to declare a namespace:

```
DECLARE ns02 CHARACTER 'http://www.ns02.com'  
.  
.  
SET OutputRoot.MRM.Element2 = InputBody.{ns02}:Element2;
```

If you use this method, you must surround the declared variable with braces to ensure that it is interpreted as a namespace.

If you are concerned that a CHARACTER variable might get changed, you can use a CONSTANT CHARACTER declaration:

```
DECLARE ns03 CONSTANT CHARACTER 'http://www.ns03.com'  
.  
.  
SET OutputRoot.MRM.Element3 = InputBody.{ns03}:Element3;
```

You can declare a namespace, constant, and variable within a module or function. However, you can declare only a namespace or constant in schema scope (that is, outside a module scope).

The Video sample provides further examples of the use of namespaces.

Namespaces are not supported by Version 2.1, so you cannot deploy a message set or message flow that uses namespaces to a Version 2.1 broker.

Querying null values in a message in the MRM domain:

If you want to compare an element to NULL, code the statement:

```
IF InputRoot.MRM.Elem2.Child1 IS NULL THEN  
  DO:  
    -- more ESQL --  
END IF;
```

If nulls are permitted for this element, this statement tests whether the element exists in the input message, or whether it exists and contains the MRM-supplied null value. The behavior of this test depends on the physical format:

- For an XML element, if the XML tag or attribute is not in the bit stream, this test returns TRUE.
- For an XML element, if the XML tag or attribute is in the bit stream and contains the MRM null value, this test returns TRUE.
- For an XML element, if the XML tag or attribute is in the bit stream and does not contain the MRM null value, this test returns FALSE.
- For a delimited TDS element, if the element has no value between the previous delimiter and its delimiter, this test returns TRUE.
- For a delimited TDS element, if the element has a value between the previous delimiter and its delimiter that is the same as the MRM-defined null value for this element, this test returns TRUE.
- For a delimited TDS element, if the element has a value between the previous delimiter and its delimiter that is not the MRM-defined null value, this test returns FALSE.
- For a CWF or fixed length TDS element, if the element's value is the same as the MRM-defined null value for this element, this test returns TRUE.
- For a CWF or fixed length TDS element, if the element's value is not the same as the MRM-defined null value, this test returns FALSE.

If you want to determine if the field is missing, rather than present but with null value, you can use the ESQL CARDINALITY function.

Setting null values in a message in the MRM domain:

To set a value of an element in an output message, you normally code an ESQL statement similar to the following:

```
SET OutputRoot.MRM.Element1.Child1 = 'xyz';
```

or its equivalent statement:

```
SET OutputRoot.MRM.Element1.Child1 VALUE = 'xyz';
```

If you set the element to a non-null value, these two statements give identical results. However, if you want to set the value to null, these two statements do not give the same result:

1. If you set the element to NULL using the following statement, the element is deleted from the message tree:

```
SET OutputRoot.MRM.Element1.Child1 = NULL;
```

The content of the output bit stream depends on the physical format:

- For an XML element, neither the XML tag or attribute nor its value are included in the output bit stream.
- For a Delimited TDS element, neither the tag (if appropriate) nor its value are included in the output bit stream. The absence of the element is typically conveyed by two adjacent delimiters.
- For a CWF or Fixed Length TDS element, the content of the output bit stream depends on whether you have set the *Default Value* property for the element. If you have set this property, the default value is included in the bit stream. If you have not set the property, an exception is raised.

This is called implicit null processing.

2. If you set the value of this element to NULL as follows:

```
SET OutputRoot.MRM.Element1.Child1 VALUE = NULL;
```

the element is not deleted from the message tree. Instead, a special value of NULL is assigned to the element. The content of the output bit stream depends on the settings of the physical format null-handling properties.

This is called explicit null processing.

Setting a complex element to NULL deletes that element and all its children.

Accessing objects in migrated message models:

If you have migrated message models that you created in WebSphere MQ Integrator Version 2.1 or WebSphere MQ Integrator Broker Version 2.1, the models created by `mqsimigratemsgsets` command include objects that you cannot create in the model in Version 5.

The following topics provide information about how to access these objects when you configure a message flow to process messages that are parsed according to those migrated models:

- “Accessing embedded simple types in migrated message models”
- “Accessing base types in migrated message models” on page 232

Accessing embedded simple types in migrated message models:

In previous releases, you could embed a simple type within a compound type in the message model. This allowed the anonymous text that can occur between the XML tags to be modeled. These simple types are referred to as embedded simple types to distinguish them from XML schema simple types. This topic is only applicable if you are working with messages that you modeled in a previous release and have imported using `mqsimigratemsgsets` command.

When an MRM message is parsed into a logical tree, embedded simple types do not have identifiers that uniquely define them in ESQL. If you want to interrogate or update the data held in an embedded simple type, you must refer to it in relation to other known objects in the message.

For example, if you want to update the embedded simple type with the text Mr. Smith, include the following ESQL in your Compute node:

```
SET OutputRoot.MRM.Person.*[3] = 'Mr.Smith';
```

This statement sets the third child of the element `Person` to `Mr.Smith`. Because this statement addresses an anonymous element in the tree (an embedded simple type that has no name), you can set its value only if you know its position in the tree.

Consider the following MRM XML message:

```
<Mess1>
  <Elem1>abc</Elem1>
  <Elem2>def<Child1>ghi</Child1></Elem2>
</Mess1>
```

You can model this XML message in the MRM using the following objects.

Message

The message *Name* property is set to `Mess1` to match the XML tag.

The *Type* property is set to `tMess1`.

Type The complex type *Name* property is set to `tMess1`.

The *Composition* property is set to `Ordered Set`.

The complex type contains the following objects:

Element

The *Name* property is set to `Elem1` to match the XML tag.

The *Type* property is set to XML Schema simple type `xsd:string`.

Element

The *Name* property is set to `Elem2` to match the XML tag.

The *Type* property is set to complex type `tElem2`.

Type The complex type *Name* property is set to `tMess2`.

The *Composition* property is set to `Sequence`.

The complex type contains the following objects:

Element

The *Name* property is set to `Child1` to match the XML tag.

The *Type* property is set to XML Schema simple type `xsd:string`.

Embedded Simple Type

`ComIbmMRM_BaseValueString`

The embedded simple type `ComIbmMRM_BaseValueString` that is embedded within `tMess2` is used to parse the data `def` from the input message. If you want to change the value of the data associated with the embedded simple type on output, code the following ESQL:

```
SET OutputRoot.MRM.Elem2.*[1] = 'xyz';
```

This generates the following output message:

```
<Mess1>
  <Elem1>abc</Elem1>
  <Elem2>xyz<Child1>ghi</Child1></Elem2>
</Mess1>
```

If you prefer not to model this message in the MRM, you can achieve the same result with the following ESQL:

```
SET OutputRoot.XML.Elem2.*[1] = 'xyz';
```

An embedded simple type does not have the facilities for null handling that is provided with elements. If you set an embedded simple type to null, it is deleted from the message tree.

In ESQL, element names are typically used to refer to and update MRM elements. The exception is when embedded simple types are present in the message. If you are using multipart messages, you must specify the message name to further qualify the embedded simple type references if the message is not the first message object in the bit stream. “Accessing embedded messages in the MRM domain” on page 227 provides further information.

Accessing base types in migrated message models:

In previous releases, you could optionally give a compound type an associated base type in the message model. This concept is provided in Version 5 by mixed content objects. This topic applies only if you are working with messages that you modeled in a previous release and have imported using `mqsimigratemsgsets` command. The base type becomes the value (data) associated with the element’s underlying complex type when the message set is imported.

If you have imported a message set that includes a compound type that has a defined base type, the migration process creates an additional child element as the first element in the corresponding complex type. The name of the additional element is automatically generated by the migration process. Although this element is displayed in the workbench, you do not need to refer to it in ESQL. You can continue to use the same ESQL statements to refer to the value of the base type, that is the name of the complex element itself.

For example, assume that you defined a compound type in Version 2.1 called `CompType1` with a base type of `STRING`, and with two children `Elem1` (`STRING`) and `Elem2` (`STRING`). You created an element `CompElem1` based on compound type `CompType1`. In ESQL you used the following statement to assign a value to the base type:

```
SET OutputRoot.MRM.CompElem1 = 'Some text value';
```

When this part of the message model is migrated to Version 5, a complex type `CompType1` is created with three elements: the original two from the Version 2.1

definition plus the additional automatically-generated element that represents the base type. You can continue to use the same statement, shown above, to assign a value to the new element. The output message generated is also identical.

Working with MRM messages and bit streams:

When you use the ASBITSTREAM function or the CREATE FIELD statement with a PARSE clause note the following points.

The ASBITSTREAM function

If you code the ASBITSTREAM function with the parser mode option set to *RootBitStream*, to parse a message tree to a bit stream, the result is an MRM document in the format specified by the message format that is built from the children of the target element in the normal way.

The target element must be a predefined message defined within the message set, or can be a self-defined message if using an XML physical format. This algorithm is identical to that used to generate the normal output bit stream. A well formed bit stream obtained in this way can be used to recreate the original tree using a CREATE statement with a PARSE clause.

If you code the ASBITSTREAM function with the parser mode option set to *FolderBitStream*, to parse a message tree to a bit stream, the generated bit stream is an MRM element built from the target element and its children. Unlike *RootBitStream* mode the target element does not have to represent a message; it can represent a predefined element within a message or self-defined element within a message.

So that the MRM parser can correctly parse the message, the path from the message to the target element within the message must be specified in the *Message Type*. The format of the path is the same as that used by message paths except that the message type prefix is not used.

For example, suppose the following message structure is used:

```
Message
  elem1
    elem11
    elem12
```

To serialize the subtree representing element *elem12* and its children, specify the message path 'message/elem1/elem12' in the *Message Type*.

If an element in the path is qualified by a namespace, specify the namespace URI between {} characters in the message path. For example if element *elem1* is qualified by namespace 'http://www.ibm.com/temp', specify the message path as 'message/{http://www.ibm.com/temp}elem1/elem12'

This mode can be used to obtain a bit stream description of arbitrary sub-trees owned by an MRM parser. When in this mode, with a physical format of XML, the XML bit stream generated is not enclosed by the 'Root Tag Name' specified for the Message in the Message Set. No XML declaration is created, even if not suppressed in the message set properties.

Bit streams obtained in this way can be used to recreate the original tree using a CREATE statement with a PARSE clause (using a mode of *FolderBitStream*).

The CREATE statement with a PARSE clause

If you code a CREATE statement with a PARSE clause, with the parser mode option set to *RootBitStream*, to parse a bit stream to a message tree, the expected bit stream is a normal MRM document. A field in the tree is created for each field in the document. This algorithm is identical to that used when parsing a bit stream from an input node

If you code a CREATE statement with a PARSE clause, with the parser mode option set to *FolderBitStream*, to parse a bit stream to a message tree, the expected bit stream is a document in the format specified by the Message Format, which is either specified directly or inherited. Unlike *RootBitStream* mode the root of the document does not have to represent an MRM message; it can represent a predefined element within a message or self-defined element within a message.

So that the MRM parser can correctly parse the message the path from the message to the target element within the message must be specified in the *Message Type*. The format of the message path is the same as that used for the ASBITSTREAM function described above.

Example of using the ASBITSTREAM function and CREATE statement with a PARSE clause in FolderBitStream mode

The following ESQL uses the message definition described above. The ESQL serializes part of the input tree using the ASBITSTREAM function, and then uses the CREATE statement with a PARSE clause to recreate the subtree in the output tree. The Input message and corresponding Output message are shown below the ESQL.

```
CREATE COMPUTE MODULE DocSampleFlow_Compute
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
  CALL CopyMessageHeaders();

  -- Set the options to be used by ASBITSTREAM and CREATE ... PARSE
  -- to be FolderBitStream and enable validation
  DECLARE parseOptions INTEGER BITOR(FolderBitStream, ValidateContent,
    ValidateValue, ValidateLocalError);

  -- Serialise the elem12 element and its children from the input bitstream
  -- into a variable
  DECLARE subBitStream BLOB
  CAST(ASBITSTREAM(InputRoot.MRM.elem1.elem12
    OPTIONS parseOptions
    SET 'DocSample'
    TYPE 'message/elem1/elem12'
    FORMAT 'XML1') AS BLOB);

  -- Set the value of the first element in the output tree
  SET OutputRoot.MRM.elem1.elem11 = 'val11';

  -- Parse the serialized sub-tree into the output tree
  IF subBitStream IS NOT NULL THEN
    CREATE LASTCHILD OF OutputRoot.MRM.elem1
      PARSE ( subBitStream
        OPTIONS parseOptions
        SET 'DocSample'
        TYPE 'message/elem1/elem12'
        FORMAT 'XML1');
  END IF;

  -- Convert the children of elem12 in the output tree to uppercase
```

```

SET OutputRoot.MRM.elem1.elem2.elem21 =
  UCASE(OutputRoot.MRM.elem1.elem2.elem21);

SET OutputRoot.MRM.elem1.elem2.elem22 =
  UCASE(OutputRoot.MRM.elem1.elem2.elem22);

  -- Set the value of the last element in the output tree
  SET OutputRoot.MRM.elem1.elem3 = 'val13';

RETURN TRUE;
END;

CREATE PROCEDURE CopyMessageHeaders() BEGIN
  DECLARE I INTEGER 1;
  DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
  WHILE I < J DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I = I + 1;
  END WHILE;
END;

END MODULE;

```

Input message :

```

<message>
  <elem1>
    <elem11>value11</elem11>
    <elem12>
      <elem121>value121</elem121>
      <elem122>value122</elem122>
    </elem12>
    <elem13>value13</elem13>
  </elem1>
</message>

```

Output message :

```

<message>
  <elem1>
    <elem11>val11</elem11>
    <elem12>
      <elem121>VALUE121</elem121>
      <elem122>VALUE122</elem122>
    </elem12>
    <elem13>val13</elem13>
  </elem1>
</message>

```

Handling large MRM messages:

When an input bit stream is parsed, and a logical tree created, the tree representation of an MRM message is typically larger, and in some cases much larger, than the corresponding bit stream. The reasons for this include:

- The addition of the pointers that link the objects together.
- Translation of character data into Unicode that can double the original size.
- The inclusion of field names that can be contained implicitly within the bit stream.
- The presence of control data that is associated with the broker's operation

Manipulation of a large message tree can, therefore, demand a great deal of storage. If you design a message flow that handles large messages made up of repeating structures, you can code specific ESQL statements that help to reduce the

storage load on the broker. These statements support both random and sequential access to the message, but assume that you do not need access to the whole message at one time.

These ESQL statements cause the broker to perform limited parsing of the message, and to keep only that part of the message tree that reflects a single record in storage at a time. If your processing requires you to retain information from record to record (for example, to calculate a total price from a repeating structure of items in an order), you can either declare, initialize, and maintain ESQL variables, or you can save values in another part of the message tree, for example `LocalEnvironment`.

This technique reduces the memory used by the broker to that needed to hold the full input and output bit streams, plus that required for one record's trees. It provides memory savings when even a small number of repeats is encountered in the message. The broker makes use of partial parsing, and the ability to parse specified parts of the message tree, to and from the corresponding part of the bit stream.

To use these techniques in your Compute node apply these general techniques:

- Copy the body of the input message as a bit stream to a special folder in the output message. This creates a modifiable copy of the input message that is not parsed and which therefore uses a minimum amount of memory.
- Avoid any inspection of the input message; this avoids the need to parse the message.
- Use a loop and a reference variable to step through the message one record at a time. For each record:
 - Use normal transforms to build a corresponding output subtree in a second special folder.
 - Use the `ASBITSTREAM` function to generate a bit stream for the output subtree that is stored in a *BitStream* element, placed in the position in the tree, that corresponds to its required position in the final bit stream.
 - Use the `DELETE` statement to delete both the current input and the output record message trees when you complete their manipulation.
 - When you complete the processing of all records, detach the special folders so that they do not appear in the output bit stream.

You can vary these techniques to suit the processing that is required for your messages. The following ESQL provides an example of one implementation, and is a rewrite of the ESQL example in “Handling large XML messages” on page 255 that uses a single `SET` statement with nested `SELECT` functions to transform a message containing nested, repeating structures.

The ESQL is dependant on a message set called `LargeMessageExample` that has been created to define messages for both the `Invoice` input format and the `Statement` output format. A message called `AllInvoices` has been created that contains a global element called `Invoice` that can repeat one or more times, and a message called `Data` that contains a global element called `Statement` that can repeat one or more times.

The definitions of the elements and attributes have been given the correct data types, therefore, the `CAST` statements used by the ESQL in the XML example are

no longer required. An XML physical format with name XML1 has been created in the message set which allows an XML message corresponding to these messages to be parsed by the MRM.

When the Statement tree is serialized using the ASBITSTREAM function the *Message Set*, *Message Type*, and *Message Format* are specified as parameters. The *Message Type* parameter contains the path from the message to the element being serialized which, in this case, is Data/Statement because the Statement element is a direct child of the Data message.

The input message to the flow is the same Invoice example message used in other parts of the documentation except that it is contained between the tags:

```
<AllInvoices> .... </AllInvoices>
```

The output message is the same as that in “Handling large XML messages” on page 255.

```
CREATE COMPUTE MODULE LargeMessageExampleFlow_Compute
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
  CALL CopyMessageHeaders();
  -- Create a special folder in the output message to hold the input tree
  -- Note : SourceMessageTree is the root element of an MRM parser
  CREATE LASTCHILD OF OutputRoot.MRM DOMAIN 'MRM' NAME 'SourceMessageTree';

  -- Copy the input message to a special folder in the output message
  -- Note : This is a root to root copy which will therefore not build trees
  SET OutputRoot.MRM.SourceMessageTree = InputRoot.MRM;

  -- Create a special folder in the output message to hold the output tree
  CREATE FIELD OutputRoot.MRM.TargetMessageTree;

  -- Prepare to loop through the purchased items
  DECLARE sourceCursor REFERENCE TO OutputRoot.MRM.SourceMessageTree.Invoice;
  DECLARE targetCursor REFERENCE TO OutputRoot.MRM.TargetMessageTree;
  DECLARE resultCursor REFERENCE TO OutputRoot.MRM;
  DECLARE grandTotal  FLOAT    0.0e0;

  -- Create a block so that it's easy to abandon processing
  ProcessInvoice: BEGIN
    -- If there are no Invoices in the input message, there is nothing to do
    IF NOT LASTMOVE(sourceCursor) THEN
      LEAVE ProcessInvoice;
    END IF;

    -- Loop through the invoices in the source tree
    InvoiceLoop : LOOP
      -- Inspect the current invoice and create a matching Statement
      SET targetCursor.Statement =
        THE (
          SELECT
            'Monthly'           AS Type,
            'Full'              AS Style,
            I.Customer.FirstName AS Customer.Name,
            I.Customer.LastName  AS Customer.Surname,
            I.Customer.Title     AS Customer.Title,
            (SELECT
              FIELDVALUE(II.Title) AS Title,
              II.UnitPrice * 1.6   AS Cost,
              II.Quantity          AS Qty
            FROM I.Purchases.Item[] AS II
            WHERE II.UnitPrice > 0.0 ) AS Purchases.Article[],
            (SELECT
              SUM( II.UnitPrice *

```

```

                II.Quantity *
                1.6
            FROM I.Purchases.Item[] AS II
            'Dollars'
            FROM sourceCursor AS I
            WHERE I.Customer.LastName <> 'White'
        );

-- Turn the current Statement into a bit stream
-- The SET parameter is set to the name of the message set
-- containing the MRM definition
-- The TYPE parameter contains the path from the from the message
-- to element being serialized
-- The FORMAT parameter contains the name of the physical format
-- name defined in the message
DECLARE StatementBitStream BLOB
    CAST(ASBITSTREAM(targetCursor.Statement
        OPTIONS FolderBitStream
        SET 'LargeMessageExample'
        TYPE 'Data/Statement'
        FORMAT 'XML1') AS BLOB);

-- If the SELECT produced a result (that is, it was not filtered
-- out by the WHERE clause), process the Statement
IF StatementBitStream IS NOT NULL THEN
    -- create a field to hold the bit stream in the result tree
    -- The Type of the element is set to MRM.BitStream to indicate
    -- to the MRM Parser that this is a bitstream
    CREATE LASTCHILD OF resultCursor
        Type MRM.BitStream
        NAME 'Statement'
        VALUE StatementBitStream;

    -- Add the current Statement's Amount to the grand total
    SET grandTotal = grandTotal + targetCursor.Statement.Amount;
END IF;

-- Delete the real Statement tree leaving only the bit stream version
DELETE FIELD targetCursor.Statement;

-- Step onto the next Invoice, removing the previous invoice and any
-- text elements that might have been interspersed with the Invoices
REPEAT
    MOVE sourceCursor NEXTSIBLING;
    DELETE PREVIOUSSIBLING OF sourceCursor;
UNTIL (FIELDNAME(sourceCursor) = 'Invoice')
    OR (LASTMOVE(sourceCursor) = FALSE)
END REPEAT;

-- If there are no more invoices to process, abandon the loop
IF NOT LASTMOVE(sourceCursor) THEN
    LEAVE InvoiceLoop;
END IF;

END LOOP InvoiceLoop;
END ProcessInvoice;

-- Remove the temporary source and target folders
DELETE FIELD OutputRoot.MRM.SourceMessageTree;
DELETE FIELD OutputRoot.MRM.TargetMessageTree;

-- Finally add the grand total
SET resultCursor.GrandTotal = grandTotal;

-- Set the output MessageType property to be 'Data'
SET OutputRoot.Properties.MessageType = 'Data';

```

```

RETURN TRUE;
END;

CREATE PROCEDURE CopyMessageHeaders() BEGIN
  DECLARE I INTEGER 1;
  DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
  WHILE I < J DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I = I + 1;
  END WHILE;
END;

END MODULE;

```

Manipulating messages in the XML domain

The following topics tell you how to deal with messages that belong to the XML domain, and that are parsed by the generic XML parser. Use this information in conjunction with the information in “Manipulating message body content” on page 171. This information is also valid for messages in the XMLNS domain, unless stated otherwise. For unique information on how to handle XMLNS messages, see “Manipulating messages in the XMLNS domain” on page 267.

An XML message can represent a complicated message model and contain a large number of different syntax elements. It is sometimes not enough to identify a field just by name and array subscript; an optional type can be associated with an element to represent some components of a message model.

The information contained in the following topics tells you how you can refer to and manipulate the elements that might occur in an XML message. It also provides information about creating new messages in a logical tree that can be successfully converted to an output bit stream. For a more detailed discussion on what each syntax element is, and how they are parsed into a message tree, see “ESQL field references” on page 792.

- “Accessing attributes in XML messages”
- “Accessing XmlDecl in an XML message” on page 241
- “Accessing DocTypeDecl in an XML message” on page 242
- “Manipulating paths and types in an XML message” on page 246
- “Ordering fields in an XML message” on page 247
- “Constructing XML output messages” on page 248
- “Transforming a simple XML message” on page 249
- “Transforming a complex XML message” on page 252
- “Handling large XML messages” on page 255
- “Returning a scalar value in an XML message” on page 258
- “Translating data in an XML message” on page 260
- “Joining data in an XML message” on page 261
- “Joining data from XML messages and database tables” on page 262
- “Working with XML messages and bit streams” on page 266

Accessing attributes in XML messages:

XML messages consist of a sequence of elements with form and content delimited by the tags. Many XML tags also include information in the form of associated attributes. The element value, and any attributes that the element might have, are treated in the tree as children of the element.

The following table lists the correlation name that you must use to refer to attributes.

Syntax element	Correlation name
Attribute	(XML.Attribute) - (XML.attr) is also supported

In the example Invoice message, the element Title within each Item element has three attributes: Category, Form, and Edition. For example, the first Title element contains:

```
<Title Category="Computer" Form="Paperback" Edition="2">The XML Companion</Title>
```

The element InputRoot.XML.Invoice.Purchases.Item[1].Title has four children in the logical tree: Category, Form, Edition, and the element value, which is The XML Companion.

If you want to access the attributes for this element, you can code the following ESQL. This extract of code retrieves the attributes from the input message and creates them as elements in the output message. It does not process the value of the element itself in this example.

```
-- Set the cursor to the first XML.Attribute of the Title, note the * after
-- (XML.Attribute) meaning any name, because the name might not be known

DECLARE cursor REFERENCE TO InputRoot.XML.Invoice.Purchases.Item[1]
        .Title.(XML.Attribute)*;

WHILE LASTMOVE(cursor) DO

    -- Create a field with the same name as the XML.Attribute and set its value
    -- to the value of the XML.Attribute

    SET OutputRoot.XML.Data.Attributes.{FIELDNAME(cursor)} = FIELDVALUE(cursor);

-- Move to the next sibling of the same TYPE to avoid the Title value
-- which is not an XML.Attribute

    MOVE cursor NEXTSIBLING REPEAT TYPE;
END WHILE;
```

When this ESQL is processed by the Compute node, the following output message is generated:

```
<Data>
  <Attributes>
    <Category>Computer</Category>
    <Form>Paperback</Form>
    <Edition>2</Edition>
  </Attributes>
</Data>
```

You can also use a SELECT statement:


```

SET OutputRoot.XML.Data.Attributes[] =
  (SELECT FIELDVALUE(I.Title)           AS title,
        FIELDVALUE(I.Title.(XML.Attribute)Category) AS category,
        FIELDVALUE(I.Title.(XML.Attribute)Form)   AS form,
        FIELDVALUE(I.Title.(XML.Attribute)Edition) AS edition
  FROM InputRoot.XML.Invoice.Purchases.Item[] AS I);

```

This generates the following output message:

```

<Data>
  <Attributes>
    <title>The XML Companion</title>
    <category>Computer</category>
    <form>Paperback</form>
    <edition>2</edition>
  </Attributes>
  <Attributes>
    <title>A Complete Guide to DB2 Universal Database</title>
    <category>Computer</category>
    <form>Paperback</form>
    <edition>2</edition>
  </Attributes>
  <Attributes>
    <title>JAVA 2 Developers Handbook</title>
    <category>Computer</category>
    <form>Hardcover</form>
    <edition>0</edition>
  </Attributes>
</Data>

```

You can qualify the SELECT with a WHERE statement to narrow down the results to obtain the same output message as the one that is generated by the WHILE statement. This second example shows that you can create the same results with less, and less complex, ESQL.

```

SET OutputRoot.XML.Data.Attributes[] =
  (SELECT FIELDVALUE(I.Title.(XML.Attribute)Category) AS category,
        FIELDVALUE(I.Title.(XML.Attribute)Form)   AS form,
        FIELDVALUE(I.Title.(XML.Attribute)Edition) AS edition
  FROM InputRoot.XML.Invoice.Purchases.Item[] AS I
  WHERE I.Title = 'The XML Companion');

```

This generates the following output message:

```

<Data>
  <Attributes>
    <Category>Computer</Category>
    <Form>Paperback</Form>
    <Edition>2</Edition>
  </Attributes>
</Data>

```

Accessing XmlDecl in an XML message:

The following table provides the correlation names for each XML syntax element in XmlDecl. Use these names to refer to the elements in input messages, and to set elements, attributes, and values in output messages.

Syntax element	Correlation name
XmlDecl	(XML.XmlDecl)
Version	(XML.Version)

Syntax element	Correlation name
Encoding	(XML."Encoding")
Standalone	(XML.Standalone)

(XML."Encoding") must include quotes, because Encoding is a reserved word.

If you want to refer to the attributes of the XML declaration in an input message, code the following ESQL. These statements are valid for a Compute node, if you are coding for a Database or Filter node, substitute Root for InputRoot.

```
IF InputRoot.XML.(XML.XmlDecl)* IS NULL THEN
  -- more ESQL --

IF InputRoot.XML.(XML.XmlDecl)*.(XML.Version)* = '1.0' THEN
  -- more ESQL --

IF InputRoot.XML.(XML.XmlDecl)*.(XML."Encoding")* = 'UTF-8' THEN
  -- more ESQL --

IF InputRoot.XML.(XML.XmlDecl)*.(XML.Standalone)* = 'no' THEN
  -- more ESQL --
```

If you want to set the XML declaration in an output message in a Compute node, code the following ESQL:

```
-- Create an XML Declaration
SET OutputRoot.XML.(XML.XmlDecl) = '';

-- Set the Version within the XML Declaration
SET OutputRoot.XML.(XML.XmlDecl).(XML.Version) = '1.0';

-- Set the Encoding within the XML Declaration
SET OutputRoot.XML.(XML.XmlDecl).(XML."Encoding") = 'UTF-8';

-- Set Standalone within the XML Declaration
SET OutputRoot.XML.(XML.XmlDecl).(XML.Standalone) = 'no';
```

This ESQL generates the following XML declaration:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

For further information on the syntax elements involved in the XML declaration, see “The XML declaration” on page 763.

Accessing DocTypeDecl in an XML message:

The XML Document Type Declaration includes the DocTypeDecl syntax element and its descendants. Together they comprise the DOCTYPE construct.

The descendants, some of which have attributes, are listed below, together with the correlation names for each XML syntax element. For more information about all these elements, see “XML document type declaration” on page 769.

Syntax element	Correlation name
AttributeDef	(XML.AttributeDef)
AttributeDefDefaultType	(XML.AttributeDefDefaultType)
AttributeDefType	(XML.AttributeDefType)

Syntax element	Correlation name
AttributeDefValue	(XML.AttributeDefValue)
AttributeList	(XML.AttributeList)
DocTypeComment	(XML.DocTypeComment)
DocTypeDecl	(XML.DocTypeDecl)
DocTypePI	(XML.DocTypePI)
DocTypeWhiteSpace	(XML.DocTypeWhiteSpace)
ElementDef	(XML.ElementDef)
EntityDecl	(XML.EntityDecl)
EntityDeclValue	(XML.EntityDeclValue)
ExternalEntityDecl	(XML.ExternalEntityDecl)
ExternalParameterEntityDecl	(XML.ExternalParameterEntityDecl)
IntSubset	(XML.IntSubset)
NotationDecl	(XML.NotationDecl)
NotationReference	(XML.NotationReference)
ParameterEntityDecl	(XML.ParameterEntityDecl)
PublicId	(XML.PublicId)
SystemId	(XML.SystemId)
UnparsedEntityDecl	(XML.UnparsedEntityDecl)

The following sections of ESQL show you how to create DocTypeDecl content in an output message generated by the Compute node. You can also use the same correlation names to interrogate all these elements within an input XML message.

The first example shows DocTypeDecl and NotationDecl:

```
-- Create a DocType Declaration named 'test'
SET OutputRoot.XML.(XML.DocTypeDecl)test = '';

-- Set a public and system ID for the DocType Declaration
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.SystemId)
= 'test.dtd';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.PublicId)
= '//this/is/a/URI/test';

-- Create an internal subset to hold our DTD definitions
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset) = '';

-- Create a Notation Declaration called 'TeX'
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.NotationDecl)TeX = '';

-- The Notation Declaration contains a SystemId and a PublicId
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.NotationDecl)TeX.(XML.SystemId) = '//TeXID';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.NotationDecl)TeX.(XML.PublicId)
= '//this/is/a/URI/TeXID';
```

The section below shows how to set up entities:

```

-- Create an Entity Declaration called 'ent1'
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.EntityDecl)ent1 = '';

-- This must contain an Entity Declaration Value
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.EntityDecl)ent1.(XML.EntityDeclValue)
  = 'this is an entity';

-- Similarly for a Parameter Entity Declaration
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.ParameterEntityDecl)ent2 = '';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.ParameterEntityDecl)ent2.(XML.EntityDeclValue)
  = '#PCDATA | sube12';

-- Create both types of External Entity, each with a
-- public and system ID
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.ExternalParameterEntityDecl)extent1 = '';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.ExternalParameterEntityDecl)extent1.(XML.SystemId)
  = 'more.txt';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.ExternalParameterEntityDecl)extent1.(XML.PublicId)
  = '//this/is/a/URI/extent1';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.ExternalEntityDecl)extent2 = '';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.ExternalEntityDecl)extent2.(XML.SystemId)
  = 'more.txt';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.ExternalEntityDecl)extent2.(XML.PublicId)
  = '//this/is/a/URI/extent2';

-- Create an Unparsed Entity Declaration called 'unpsd'
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.UnparsedEntityDecl)unpsd = '';
-- This has a SystemId, PublicId and Notation Reference
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.UnparsedEntityDecl).(XML.SystemId) = 'me.gif';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.UnparsedEntityDecl).(XML.PublicId)
  = '//this/is/a/URI/me.gif';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.UnparsedEntityDecl).(XML.NotationReference) = 'TeX';

```

The section below shows DocTypeWhiteSpace, DocTypeProcessingInstruction, and DocTypeComment:

```

-- Create some whitespace in the DocType Declaration
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.DocTypeWhiteSpace) = '      ';

-- Create a Processing Instruction named 'test'
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.DocTypePI)test = 'Do this';

-- Add a DocTypeComment
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
  (XML.DocTypeComment) = 'this is a comment';

```

The section below shows how to set up elements:

```

-- Create a variety of Elements

SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.ElementDef)subel2 = '#PCDATA';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.ElementDef)subel1 = '(subel2 | e14)*';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.ElementDef)e11 = '#PCDATA';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.ElementDef)e12 = '#PCDATA | subel2)*';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.ElementDef)e13 = '#PCDATA | subel2)*';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.ElementDef)e14 = '#PCDATA';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.ElementDef)e15 = '#PCDATA | subel1)*';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.ElementDef)e16 = '#PCDATA';

```

The section below shows how to set up attribute lists:

```

-- Create an AttributeList for element subel1

SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.AttributeList)subel1 = '';

-- Create an attribute called 'size' with enumerated
-- values 'big' or 'small'
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.AttributeList)subel1.(XML.AttributeDef)size = '';

SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.AttributeList)subel1.(XML.AttributeDef)size.
(XML.AttributeDefType) = '(big | small)';

-- Set the default value of our attribute to be 'big'
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.AttributeList)subel1.(XML.AttributeDef)size.
(XML.AttributeDefValue) = 'big';

-- Create another attribute - this time we specify
-- the DefaultType as being #REQUIRED
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.AttributeList)subel1.(XML.AttributeDef)shape = '';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.AttributeList)subel1.(XML.AttributeDef)shape.
(XML.AttributeDefType) = '(round | square)';

-- Create another attribute list for element e15 with
-- one attribute, containing CDATA which is #IMPLIED
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.AttributeList)e15 = '';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.AttributeList)e15.(XML.AttributeDef)e15satt = '';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.AttributeList)e15.(XML.AttributeDef)e15satt.
(XML.AttributeDefType)CDATA = '';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.IntSubset).
(XML.AttributeList)e15.(XML.AttributeDef)e15satt.
(XML.AttributeDefDefaultType) = 'IMPLIED';

```

This generates the following DocType Declaration (note that carriage returns have been added for ease of viewing):

```

<!DOCTYPE test PUBLIC "-//this/is/a/URI/test" "test.dtd"
[<!NOTATION TeX PUBLIC "-//this/is/a/URI/TeXID" "//TeXID">
<!ENTITY ent1 "this is an entity">
<!ENTITY % ent2 "#PCDATA | sube12">
<!ENTITY % extent1 PUBLIC "-//this/is/a/URI/extent1" "more.txt">
<!ENTITY extent2 PUBLIC "-//this/is/a/URI/extent2" "more.txt">
<!ENTITY unpsd PUBLIC "-//this/is/a/URI/me.gif" "me.gif" NDATA TeX> <?test Do this?>
<!--this is a comment-->
<!ELEMENT sube12 (#PCDATA)>
<!ELEMENT sube11 (sube12 | e14)+>
<!ELEMENT e11 (#PCDATA)>
<!ELEMENT e12 (#PCDATA | sube12)*>
<!ELEMENT e13 (#PCDATA | sube12)*>
<!ELEMENT e14 (#PCDATA)>
<!ELEMENT e15 (#PCDATA | sube11)*>
<!ELEMENT e16 (#PCDATA)>
<!ATTLIST sube11
    size (big | small) "big"
    shape (round | square) #REQUIRED>
<!ATTLIST e15
    e15satt CDATA #IMPLIED>
]>

```

Manipulating paths and types in an XML message:

When you refer to or set elements within an XML message body, you must use the correct correlation names, in ESQL field references, to address them. The following table lists the correlation names for all valid elements. For correlation names for attributes XmlDec, and DocTypeDecl, see “Accessing attributes in XML messages” on page 239, “Accessing XmlDecl in an XML message” on page 241, and “Accessing DocTypeDecl in an XML message” on page 242. For information about field references, see “ESQL field references” on page 792.

Syntax element	Correlation name
CDataSection	(XML.CDataSection)
Comment	(XML.Comment)
Content	(XML.Content) - (XML.pCDATA) is also supported
Element	(XML.Element) - (XML.tag) is also supported
EntityReferenceEnd	(XML.EntityReferenceEnd)
EntityReferenceStart	(XML.EntityReferenceStart)
ProcessingInstruction	(XML.ProcessingInstruction)
WhiteSpace	(XML.WhiteSpace)

When a type is not present in a path element, the type of the syntax element is not important. That is, a path element of name matches any syntax element with the name of name, regardless of the element type. In the same way that a path element can specify a name and not a type, a path element can specify a type and not a name. This type of path element matches any syntax element that has the specified type, regardless of name. The following is an example of this:

```
FIELDNAME(InputBody.(XML.Element)[1])
```

This example returns the name of the first element in the body of the message. The following example of generic XML shows when it is necessary to use types in paths:

```
<tag1 attr1='abc'>
  <attr1>123<attr1>
</tag1>
```

The path `InputBody.tag1.attr1` refers to the attribute called `attr1`, because attributes appear before nested elements in a syntax tree generated by an XML parser. To refer to the element called `attr1` you must use a path:

```
InputBody.tag1.(XML.Element)attr1
```

It is always advisable to include types in these cases to be explicit about which syntax element is being referred to.

The following ESQL:

```
SET OutputRoot.XML.Element1.(XML.Element)Attribute1 = '123';
```

is essentially shorthand for the following, fully-qualified path :

```
SET OutputRoot.XML.(XML.Element)Element1.(XML.Element)Attribute1.
(XML.Content) = '123';
```

Consider the following XML:

```
<?xml version="1.0"?>
<!DOCTYPE Order SYSTEM "Order.dtd">
<Order>
  <ItemNo>1</ItemNo>
  <Quantity>2</Quantity>
</Order>
```

The path `InputBody.Order` refers to the `(XML.DocTypeDecl)` syntax element, because this appears before the XML Body in the syntax tree and has the same name. To refer to the element `ItemNo` you need to use a path `InputBody.(XML.Element)Order.ItemNo`. The following example demonstrates the same idea, using the following XML input message:

```
<doc><i1>100</i1></doc>
```

To assign 112233 to `<i1>`, you must use the following ESQL expression:

```
SET OutputRoot.XML.(XML.Element)doc.I1=112233;
```

Ordering fields in an XML message:

When you create an XML output message in a Compute node, the order in which your lines of ESQL appear is important, because the message elements are created in the order in which you code them.

Consider the following XML message:

```
<Order>
  <ItemNo>1</ItemNo>
  <Quantity>2</Quantity>
</Order>
```

If you want to add a DocType Declaration to this, insert the DocType Declaration before you copy the input message to the output message. For example:

```

SET OutputRoot.XML.(XML.XmlDecl) = '';
SET OutputRoot.XML.(XML.XmlDecl).(XML.Version) = '1.0';
SET OutputRoot.XML.(XML.DocTypeDecl)Order = '';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.SystemId)
  = 'NewDtdName.dtd';

SET OutputRoot = InputRoot;
-- more ESQL --

```

If you put the last statement to copy the input message before the XML-specific statements, the following XML is generated for the output message. This is not well-formed and fails when written from the message tree to a bit stream in the output node:

```

<Order>
  <ItemNo>1</ItemNo>
  <Quantity>2</Quantity>
</Order>
<?xml version="1.0"?>
<!DOCTYPE Order SYSTEM "Order.dtd">

```

Constructing XML output messages:

Within the Compute node, you can create output XML messages by taking information from an input message (which might or might not be XML), from a database, or from other information or calculations. In addition to the general guidance provided in “Manipulating message body content” on page 171, consider the following points:

- You might want an empty element in the output message. On input, an empty element of the form `<tag></tag>` is interpreted as identical to one of the form `<tag/>`. On output, the default behavior of the generic XML parser is to generate empty elements in the first of these two forms. If you require the second (short) form of empty element, set the content of the element to NULL. The statement:

```
SET OutputRoot.XML.Invoice.Cashier.(XML.Content) = NULL;
```

generates the following XML:

```
<Invoice><Cashier/></Invoice>
```

- It is possible to code ESQL that creates invalid XML or element content:
 - If you code ESQL that creates an XML message that is not well formed (that is, compliant with the XML specification), the generic XML parser invoked by the output node or nodes in the message flow to create an output bit stream from the logical message tree cannot do so.

An example of badly-formed XML is shown below, where the ESQL constructs two top-level tags:

```
SET OutputRoot.XML.Element1 = 'a';
SET OutputRoot.XML.Element2 = 'b';
```

It is possible to create a message tree that, when parsed, results in tags that are written as attributes, attributes that are written as tags, and tags that are not written at all. This might happen, for example, if you copy elements to the output message from an input message that is not an XML message.

It is also possible to create a message in which the contents are not in the expected order; this is further described in “Ordering fields in an XML message” on page 247.

If your message flow does not create an output message successfully, or the output message does not have the content that you expect, check the ESQL code that you have written to create the output message in the Compute node.

- In addition to ensuring that the structure of the XML message tree is valid, you must also ensure that the values written into the fields are valid. Because character-by-character validation is not performed by the parser when it constructs an XML message bit stream from the message tree, you can write invalid characters into the output XML message. This might result in an output message that cannot be parsed, or is parsed incorrectly with respect to the structure (tags and attributes) and the content.

You might want to include a test on the data values that you insert into the output message, or use the CAST function.

Transforming a simple XML message:

When you code the ESQL for a Compute node, use the SELECT statement to transform simple messages.

Examples

Review the following examples and modify them for your own use. They are all based on the Invoice message as input.

Consider the following ESQL:

```
SET OutputRoot.XML.Data.Output[] =  
  (SELECT R.Quantity, R.Author FROM InputRoot.XML.Invoice.Purchases.Item[] AS R);
```

When the Invoice message is processed by this ESQL, the following output message is produced:

```
<Data>  
  <Output>  
    <Quantity>2</Quantity>  
    <Autho>Neil Bradley</Autho>  
  </Output>  
  <Output>  
    <Quantity>1</Quantity>  
    <Autho>Don Chamberlin</Autho>  
  </Output>  
  <Output>  
    <Quantity>1</Quantity>  
    <Autho>Philip Heller, Simon Roberts</Autho>  
  </Output>  
</Data>
```

There are three Output fields, one for each Item field. This is because, by default, SELECT creates an item in its result list for each item described by its FROM list. Within each Output field, there is a Field for each field named in the SELECT clause and these are in the order in which they are specified within the SELECT, not in the order in which they appear in the incoming message.

The R introduced by the final AS keyword is known as a correlation name. It is a local variable that represents in turn each of the fields addressed by the FROM clause. There is no significance to the name chosen. In summary, this simple transform does two things:

1. It discards unwanted fields.
2. It guarantees the order of the fields.

Here is the same transform implemented by a procedural algorithm:

```
DECLARE i INTEGER 1;
DECLARE count INTEGER CARDINALITY(InputRoot.XML.Invoice.Purchases.Item[]);

WHILE (i <= count)
  SET OutputRoot.XML.Data.Output[i].Quantity = InputRoot.XML.Invoice.Purchases.Item[i].Quantity;
  SET OutputRoot.XML.Data.Output[i].Author = InputRoot.XML.Invoice.Purchases.Item[i].Author;
  SET i = i+1;
END WHILE;
```

These examples show that the SELECT version of the transform is much more concise. It also executes faster.

The following example shows a more advanced transformation:

```
SET OutputRoot.XML.Data.Output[] =
  (SELECT R.Quantity AS Book.Quantity,
         R.Author AS Book.Author
   FROM InputRoot.XML.Invoice.Purchases.Item[] AS R
  );
```

In this transform, there is an AS clause associated with each item in the SELECT clause. This gives each field in the result an explicit name rather than the field names being inherited from the input. These names can be paths (that is, a dot separated list of names), as shown in the example. Thus, the output message's structure can be arbitrarily different from the input message's. Using the same Invoice message, the result is:

```
<Data>
  <Output>
    <Book>
      <Quantity>2</Quantity>
      <Author>Neil Bradley</Author>
    </Book>
  </Output>
</Data>

<Data>
  <Output>
    <Book>
      <Quantity>2</Quantity>
      <Author>Neil Bradley</Author>
    </Book>
  </Output>
  <Output>
    <Book>
      <Quantity>1</Quantity>
      <Author>Don Chamberlin</Author>
    </Book>
  </Output>
  <Output>
    <Book>
      <Quantity>1</Quantity>
      <Author>Philip Heller, Simon Roberts</Author>
    </Book>
  </Output>
</Data>
```

The expressions in the SELECT clause can be of any complexity and there are no special restrictions. They can include operators, functions, literals, and they can

refer to variables or to fields not related to the correlation name. The following example shows more complex expressions:

```
SET OutputRoot.XML.Data.Output[] =
  (SELECT 'Start' AS Header,
         'Number of books:' || R.Quantity AS Book.Quantity,
         R.Author || ':Name and Surname' AS Book.Author,
         'End' AS Trailer
   FROM InputRoot.XML.Invoice.Purchases.Item[] AS R
  );
```

Using the same Invoice message, the result in this case is:

```
<Data>
  <Output>
    <Header>Start</Header>
    <Book>
      <Quantity>Number of books:2</Quantity>
      <Author>Neil Bradley:Name and Surname</Author>
    </Book>
    <Trailer>End</Trailer>
  </Output>
  <Output>
    <Header>Start</Header>
    <Book>
      <Quantity>Number of books:1</Quantity>
      <Author>Don Chamberlin:Name and Surname</Author>
    </Book>
    <Trailer>End</Trailer>
  </Output>
  <Output>
    <Header>Start</Header>
    <Book>
      <Quantity>Number of books:1</Quantity>
      <Author>Philip Heller, Simon Roberts:Name and Surname</Author>
    </Book>
    <Trailer>End</Trailer>
  </Output>
</Data>
```

As shown above, the AS clauses of the SELECT clause contain a path that describes the full name of the field to be created in the result. These paths can also specify (as is normal for paths) the type of field to be created. The following example transform specifies the field types. In this case, XML tagged data is transformed to XML attributes:

```
SET OutputRoot.XML.Data.Output[] =
  (SELECT R.Quantity.* AS Book.(XML.Attribute)Quantity,
         R.Author.* AS Book.(XML.Attribute)Author
   FROM InputRoot.XML.Invoice.Purchases.Item[] AS R
  );
```

Using the same Invoice message, the result is:

```

<Data>
  <Output>
    <Book Quantity="2" Author="Neil Bradley"/>
  </Output>
  <Output>
    <Book Quantity="1" Author="Don Chamberlin"/>
  </Output>
  <Output>
    <Book Quantity="1" Author="Philip Heller, Simon Roberts"/>
  </Output>
</Data>

```

Finally, you can use a *WHERE* clause to eliminate some of the results. In the following example a *WHERE* clause is used to remove results in which a specific criterion is met. An entire result is either included or excluded:

```

SET OutputRoot.XML.Data.Output[] =
  (SELECT R.Quantity AS Book.Quantity,
         R.Author AS Book.Author
   FROM InputRoot.XML.Invoice.Purchases.Item[] AS R
   WHERE R.Quantity = 2
  );

```

Using the same input message, the result is:

```

<Data>
  <Output>
    <Book>
      <Quantity>2</Quantity>
      <Author>Neil Bradley</Author>
    </Book>
  </Output>
</Data>

```

Transforming a complex XML message:

When you code the ESQL for a Compute node, use the *SELECT* statement for complex message transformation.

Examples

Review the following examples and modify them for your own use. They are all based on the Invoice message as input:

In this example, Invoice contains a variable number of Items. The transform is shown below:

```

SET OutputRoot.XML.Data.Statement[] =
  (SELECT I.Customer.Title AS Customer.Title,
        I.Customer.FirstName || ' ' || I.Customer.LastName AS Customer.Name,
        COALESCE(I.Customer.PhoneHome, '') AS Customer.Phone,
        (SELECT II.Title AS Desc,
          CAST(II.UnitPrice AS FLOAT) * 1.6 AS Cost,
          II.Quantity AS Qty
        FROM I.Purchases.Item[] AS II
        WHERE II.UnitPrice > 0.0
        ) AS Purchases.Article[],
        (SELECT SUM( CAST(II.UnitPrice AS FLOAT) *
          CAST(II.Quantity AS FLOAT) *
          1.6
        )
        FROM I.Purchases.Item[] AS II
        ) AS Amount,
        'Dollars' AS Amount.(XML.Attribute)Currency

  FROM InputRoot.XML.Invoice[] AS I
  WHERE I.Customer.LastName <> 'Brown'
);

```

The output message that is generated is:

```

<Data>
<Statement>
<Customer>
<Title>Mr</Title>
<Name>Andrew Smith</Name>
<Phone>01962818000</Phone>
</Customer>
<Purchases>
<Article>
<Desc Category="Computer" Form="Paperback" Edition="2">The XML Companion</Desc>
<Cost>4.472E+1</Cost>
<Qty>2</Qty>
</Article>
<Article>
<Desc Category="Computer" Form="Paperback" Edition="2">
  A Complete Guide to DB2 Universal Database</Desc>
<Cost>6.872E+1</Cost>
<Qty>1</Qty>
</Article>
<Article>
<Desc Category="Computer" Form="Hardcover" Edition="0">JAVA 2 Developers Handbook</Desc>
<Cost>9.5984E+1</Cost>
<Qty>1</Qty>
</Article>
</Purchases>
<Amount Currency="Dollars">2.54144E+2</Amount>
</Statement>
</Data>

```

This transform has two SELECTs nested inside each other. The outer one operates on the list of Invoices. The inner one operates on the list of Items. The AS clause associated with the inner SELECT expects an array:

```

(SELECT II.Title AS Desc,
      CAST(II.UnitPrice AS FLOAT) * 1.6 AS Cost,
      II.Quantity AS Qty
FROM I.Purchases.Item[] AS II
WHERE II.UnitPrice > 0.0
)
-- Note the use of [] in the next expression
AS Purchases.Article[],

```

This tells the outer select to expect a variable number of Items in each result. Each SELECT has its own correlation name: I for the outer select and II for the inner

one. Each SELECT typically uses its own correlation name, but the inner SELECT's FROM clause refers to the outer SELECT's correlation name:

```
(SELECT II.Title AS Desc,
      CAST(II.UnitPrice AS FLOAT) * 1.6 AS Cost,
      II.Quantity AS Qty
-- Note the use of I.Purchases.Item in the next expression
FROM I.Purchases.Item[] AS II
WHERE II.UnitPrice > 0.0 ) AS Purchases.Article[],
```

This tells the inner SELECT to work with the current Invoice's Items. Both SELECTs contain WHERE clauses. The outer one uses one criterion to discard certain Customers and the inner one uses a different criterion to discard certain Items. The example also shows the use of COALESCE to prevent missing input fields causing the corresponding output field to be missing. Finally, it also uses the column function SUM to add together the value of all Items in each Invoice. Column functions are discussed in "Referencing columns in a database" on page 205.

When the fields Desc are created, the whole of the input Title field is copied: the XML attributes and the field value. If you do not want these attributes in the output message, you can use the FIELDVALUE function to discard them; for example code the following ESQL:

```
SET OutputRoot.XML.Data.Statement[] =
  (SELECT I.Customer.Title AS Customer.Title,
        I.Customer.FirstName || ' ' || I.Customer.LastName AS Customer.Name,
        COALESCE(I.Customer.PhoneHome, '') AS Customer.Phone,
        (SELECT FIELDVALUE(II.Title) AS Desc,
          CAST(II.UnitPrice AS FLOAT) * 1.6 AS Cost,
          II.Quantity AS Qty
        FROM I.Purchases.Item[] AS II
        WHERE II.UnitPrice > 0.0 ) AS Purchases.Article[],
        (SELECT SUM( CAST(II.UnitPrice AS FLOAT) *
          CAST(II.Quantity AS FLOAT) *
          1.6 )
        FROM I.Purchases.Item[] AS II ) AS Amount,
        'Dollars' AS Amount.(XML.Attribute)Currency

  FROM InputRoot.XML.Invoice[] AS I
  WHERE I.Customer.LastName <> 'Brown'
  );
```

That generates the following output message:

```

<Data>
  <Statement>
    <Customer>
      <Title>Mr</Title>
      <Name>Andrew Smith</Name>
      <Phone>01962818000</Phone>
    </Customer>
    <Purchases>
      <Article>
        <Desc>The XML Companion</Desc>
        <Cost>4.472E+1</Cost>
        <Qty>2</Qty>
      </Article>
      <Article>
        <Desc>A Complete Guide to DB2 Universal Database</Desc>
        <Cost>6.872E+1</Cost>
        <Qty>1</Qty>
      </Article>
      <Article>
        <Desc>JAVA 2 Developers Handbook</Desc>
        <Cost>9.5984E+1</Cost>
        <Qty>1</Qty>
      </Article>
    </Purchases>
    <Amount Currency="Dollars">2.54144E+2</Amount>
  </Statement>
</Data>

```

Handling large XML messages:

When an input bit stream is parsed, and a logical tree created, the tree representation of an XML message is typically bigger, and in some cases much bigger, than the corresponding bit stream. The reasons for this include:

- The addition of the pointers that link the objects together
- Translation of character data into Unicode; this can double the size
- The inclusion of field names that might have been implicit in the bit stream
- The presence of control data that is associated with the broker's operation

Manipulating a large message tree can demand a lot of storage. If you design a message flow that handles large messages made up of repeating structures, you can code ESQL statements that help to reduce the storage load on the broker. These statements support both random and sequential access to the message, but assume that you do not need access to the whole message at one time.

These ESQL statements cause the broker to perform limited parsing of the message, and to keep only that part of the message tree that reflects a single record in storage at a time. If your processing requires you to retain information from record to record (for example, to calculate a total price from a repeating structure of items in an order), you can either declare, initialize, and maintain ESQL variables, or you can save values in another part of the message tree, for example `LocalEnvironment`.

This technique reduces the memory used by the broker to that needed to hold the full input and output bit streams, plus that needed for just one record's trees, and provides memory savings when even a small number of repeats is encountered in the message. The broker uses partial parsing and the ability to parse specified parts of the message tree to and from the corresponding part of the bit stream.

To use these techniques in your Compute node, apply these general techniques:

- Copy the body of the input message as a bit stream to a special folder in the output message. This creates a modifiable copy of the input message that is not parsed and that therefore uses a minimum amount of memory.
- Avoid any inspection of the input message. This avoids the need to parse the message.
- Use a loop and a reference variable to step through the message one record at a time. For each record:
 - Use normal transforms to build a corresponding output subtree in a second special folder.
 - Use the ASBITSTREAM function to generate a bit stream for the output subtree that is stored in a BitStream element placed in the position in the tree that corresponds to its required position in the final bit stream.
 - Use the DELETE statement to delete both the current input and output record message trees when you have completed their manipulation.
 - When you have completed the processing of all records, detach the special folders so that they do not appear in the output bit stream.

You can vary these techniques to suit the processing required for your messages. The ESQL below provides an example of one implementation, and is a rewrite of the ESQL example in “Transforming a complex XML message” on page 252. It uses a single SET statement with nested SELECT functions to transform a message containing nested, repeating structures.


```

-- Copy the MQMD header
SET OutputRoot.MQMD = InputRoot.MQMD;

-- Create a special folder in the output message to hold the input tree
-- Note : SourceMessageTree is the root element of an XML parser
CREATE LASTCHILD OF OutputRoot.XML.Data DOMAIN 'XML' NAME 'SourceMessageTree';

-- Copy the input message to a special folder in the output message
-- Note : This is a root to root copy which will therefore not build trees
SET OutputRoot.XML.Data.SourceMessageTree = InputRoot.XML;

-- Create a special folder in the output message to hold the output tree
CREATE FIELD OutputRoot.XML.Data.TargetMessageTree;

-- Prepare to loop through the purchased items
DECLARE sourceCursor REFERENCE TO OutputRoot.XML.Data.SourceMessageTree.Invoice;
DECLARE targetCursor REFERENCE TO OutputRoot.XML.Data.TargetMessageTree;
DECLARE resultCursor REFERENCE TO OutputRoot.XML.Data;
DECLARE grandTotal FLOAT 0.0e0;

-- Create a block so that it's easy to abandon processing
ProcessInvoice: BEGIN
-- If there are no Invoices in the input message, there is nothing to do
IF NOT LASTMOVE(sourceCursor) THEN
    LEAVE ProcessInvoice;
END IF;

-- Loop through the invoices in the source tree
InvoiceLoop : LOOP
-- Inspect the current invoice and create a matching Statement
SET targetCursor.Statement =
    THE (
        SELECT
            'Monthly' AS (XML.Attribute)Type,
            'Full' AS (0x03000000)Style[1],
            I.Customer.FirstName AS Customer.Name,
            I.Customer.LastName AS Customer.Surname,
            I.Customer.Title AS Customer.Title,
            (SELECT
                FIELDVALUE(II.Title) AS Title,
                CAST(II.UnitPrice AS FLOAT) * 1.6 AS Cost,
                II.Quantity AS Qty
            FROM I.Purchases.Item[] AS II
            WHERE II.UnitPrice > 0.0 ) AS Purchases.Article[],
            (SELECT
                SUM( CAST(II.UnitPrice AS FLOAT) *
                    CAST(II.Quantity AS FLOAT) *
                    1.6 )
            FROM I.Purchases.Item[] AS II ) AS Amount,
            'Dollars' AS Amount.(XML.Attribute)Currency
        FROM sourceCursor AS I
        WHERE I.Customer.LastName <> 'White'
    );

-- Turn the current Statement into a bit stream
DECLARE StatementBitStream BLOB
    CAST(ASBITSTREAM(targetCursor.Statement OPTIONS FolderBitStream) AS BLOB);

-- If the SELECT produced a result (that is, it was not filtered out by the WHERE
-- clause), process the Statement
IF StatementBitStream IS NOT NULL THEN
-- create a field to hold the bit stream in the result tree
CREATE LASTCHILD OF resultCursor
    Type XML.BitStream
    NAME 'StatementBitStream'
    VALUE StatementBitStream;

-- Add the current Statement's Amount to the grand total
-- Note that the cast is necessary because of the behavior of the XML syntax element
SET grandTotal = grandTotal + CAST(targetCursor.Statement.Amount AS FLOAT);
END IF;

-- Delete the real Statement tree leaving only the bit stream version

```

This produces the following output message:

```
<Data>
<Statement Type="Monthly" Style="Full">
  <Customer>
    <Name>Andrew</Name>
    <Surname>Smith</Surname>
    <Title>Mr</Title>
  </Customer>
  <Purchases>
    <Article>
      <Title>The XML Companion </Title>
      <Cost>4.472E+1</Cost>
      <Qty>2</Qty>
    </Article>
    <Article>
      <Title>A Complete Guide to DB2 Universal Database</Title>
      <Cost>6.872E+1</Cost>
      <Qty>1</Qty>
    </Article>
    <Article>
      <Title>JAVA 2 Developers Handbook</Title>
      <Cost>9.5984E+1</Cost>
      <Qty>1</Qty>
    </Article>
  </Purchases>
  <Amount Currency="Dollars">2.54144E+2</Amount>
</Statement>
<GrandTotal>2.54144E+2</GrandTotal>
</Data>
```

Returning a scalar value in an XML message:

Use a SELECT statement to return a scalar value by including both the THE and ITEM keywords, for example:

```
1 + THE(SELECT ITEM T.a FROM Body.Test.A[] AS T WHERE T.b = '123')
```

Use of the ITEM keyword:

The following example shows the use of the ITEM keyword to select one item and create a single value.

```
SET OutputRoot.MQMD = InputRoot.MQMD;
SET OutputRoot.XML.Test.Result[] =
  (SELECT ITEM T.UnitPrice FROM InputBody.Invoice.Purchases.Item[] AS T);
```

When the Invoice message is received as input, the ESQL shown generates the following output message:

```
<Test>
  <Result>27.95</Result>
  <Result>42.95</Result>
  <Result>59.99</Result>
</Test>
```

When the ITEM keyword is specified, the output message includes a list of scalar values. Compare this message to the one that is produced if the ITEM keyword is omitted, in which a list of fields (name-value pairs) is generated:

```

<Test>
  <Result>
    <UnitPrice>27.95</UnitPrice>
  </Result>
  <Result>
    <UnitPrice>42.95</UnitPrice>
  </Result>
  <Result>
    <UnitPrice>59.99</UnitPrice>
  </Result>
</Test>

```

Effects of the THE keyword:

The THE keyword converts a list containing one item to the item itself.

The two previous examples both specified a list as the source of the SELECT in the FROM clause (the field reference has [] at the end to indicate an array), so typically the SELECT generates a list of results. Because of this you need to specify a list as the target of the assignment (thus the "Result[]" as the target of the assignment). However, you often know that the WHERE clause that you specify as part of the SELECT only returns TRUE for one item in the list. In this case use the THE keyword.

The following example shows the effect of using the THE keyword:

```

SET OutputRoot.MQMD = InputRoot.MQMD;

SET OutputRoot.XML.Test.Result =
  THE (SELECT T.Publisher, T.Author FROM InputBody.Invoice.Purchases.Item[]
      AS T WHERE T.UnitPrice = 42.95);

```

The THE keyword means that the target of the assignment becomes OutputRoot.XML.Test.Result (the "[]" is not permitted). Its use generates the following output message:

```

<Test>
  <Result>
    <Publisher>Morgan Kaufmann Publishers</Publisher>
    <Author>Don Chamberlin</Author>
  </Result>
</Test>

```

Selecting from a list of scalars:

Consider the following sample input message:

```

<Test>
  <A>1</A>
  <A>2</A>
  <A>3</A>
  <A>4</A>
  <A>5</A>
</Test>

```

If you code the following ESQL statements to process this message:

```

SET OutputRoot.XML.Test.A[] =
  (SELECT ITEM A from InputBody.Test.A[]
   WHERE CAST(A AS INTEGER) BETWEEN 2 AND 4);

```

the following output message is generated:

```
<A>2</A>
<A>3</A>
<A>4</A>
```

Translating data in an XML message:

You often need to translate data from one form to another. For example, in one message the types of items are known by names and in another message the items are known by numbers. For example:

Type Name	Type Code
Confectionary	2000
Newspapers	3000
Hardware	4000

Consider the following input message:

```
<Data>
  <Items>
    <Item>
      <Cat>1000</Cat>
      <Description>Milk Chocolate Bar</Description>
      <Type>Confectionary</Type>
    </Item>
    <Item>
      <Cat>1001</Cat>
      <Description>Daily Newspaper</Description>
      <Type>NewsPapers</Type>
    </Item>
    <Item>
      <Cat>1002</Cat>
      <Description>Kitchen Sink</Description>
      <Type>Hardware</Type>
    </Item>
  </Items>
  <TranslateTable>
    <Translate>
      <Name>Confectionary</Name>
      <Number>2000</Number>
    </Translate>
    <Translate>
      <Name>NewsPapers</Name>
      <Number>3000</Number>
    </Translate>
    <Translate>
      <Name>Hardware</Name>
      <Number>4000</Number>
    </Translate>
  </TranslateTable>
</Data>
```

This message has two sections: the first is a list of items in which each item has a catalogue number and a type; the second is a translate table between descriptive type names and numeric type codes. If you include a Compute node with the following transform:

```

SET OutputRoot.XML.Result.Items.Item[] =
  (SELECT M.Cat, M.Description, T.Number As Type
   FROM
     InputRoot.XML.Data.Items.Item[] As M,
     InputRoot.XML.Data.TranslateTable.Translate[] As T
   WHERE M.Type = T.Name
  );

```

the following output message is generated:

```

<Result>
  <Items>
    <Item>
      <Cat>1000</Cat>
      <Description>Milk Chocolate Bar</Description>
      <Type>2000</Type>
    </Item>
    <Item>
      <Cat>1001</Cat>
      <Description>Daily Newspaper</Description>
      <Type>3000</Type>
    </Item>
    <Item>
      <Cat>1002</Cat>
      <Description>Kitchen Sink</Description>
      <Type>4000</Type>
    </Item>
  </Items>
</Result>

```

In the result, each type name has been converted to its corresponding code. In this example, both the data and the translate table were in the same message tree, although this is not a requirement. For example, the translate table could be coded in a database, or might have been set up in LocalEnvironment by a previous Compute node.

Joining data in an XML message:

The FROM clause is not restricted to having one item. Specifying multiple items in the FROM clause produces the usual Cartesian product joining effect, in which there is an item in the result for all combinations of items in the two lists. This is the same joining effect as standard SQL.

The Invoice message includes a set of customer details, payment details, and details of the purchases that the customer makes. If you code the following ESQL to process the input Invoice message:

```

SET OutputRoot.XML.Items.Item[] =
  (SELECT D.LastName, D.Billing,
         P.UnitPrice, P.Quantity
   FROM InputBody.Invoice.Customer[] AS D,
         InputBody.Invoice.Purchases.Item[] AS P);

```

the following output message is generated:

```

<Items>
  <Item>
    <LastName>Smith</LastName>
    <Billing>
      <Address>14 High Street</Address>
      <Address>Hursley Village</Address>
      <Address>Hampshire</Address>
      <PostCode>S0213JR</PostCode>
    </Billing>
    <UnitPrice>27.95</UnitPrice>
    <Quantity>2</Quantity>
  </Item>
  <Item>
    <LastName>Smith</LastName>
    <Billing>
      <Address>14 High Street</Address>
      <Address>Hursley Village</Address>
      <Address>Hampshire</Address>
      <PostCode>S0213JR</PostCode>
    </Billing>
    <UnitPrice>42.95</UnitPrice>
    <Quantity>1</Quantity>
  </Item>
  <Item>
    <LastName>Smith</LastName>
    <Billing>
      <Address>14 High Street</Address>
      <Address>Hursley Village</Address>
      <Address>Hampshire</Address>
      <PostCode>S0213JR</PostCode>
    </Billing>
    <UnitPrice>59.99</UnitPrice>
    <Quantity>1</Quantity>
  </Item>
</Items>

```

There are three results, giving the number of descriptions in the first list (one) multiplied by the number of prices in the second (three). The results systematically work through all the combinations of the two lists. You can see this by looking at the LastName and UnitPrice fields selected from each result:

```

LastName Smith   UnitPrice 27.95
LastName Smith   UnitPrice 42.95
LastName Smith   UnitPrice 59.99

```

You can join data that occurs in a list and a non-list, or in two non-lists, and so on. For example:

```

OutputRoot.XML.Test.Result1[] =
  (SELECT ... FROM InputBody.Test.A[], InputBody.Test.b);
OutputRoot.XML.Test.Result1 =
  (SELECT ... FROM InputBody.Test.A, InputBody.Test.b);

```

Note the location of the [] in each case. Any number of items can be specified in the FROM list, not just one or two. If any of the items specify [] to indicate a list of items, the SELECT generates a list of results (the list might contain only one item, but the SELECT can potentially return a list of items). The target of the assignment must specify a list (so must end in [] or you must use the THE keyword if you know that the WHERE clause guarantees that only one combination is matched).

Joining data from XML messages and database tables:

You can use SELECT statements that interact with both message data and databases. You can also nest a SELECT that interacts with one type of data within a SELECT that interacts with the other type.

Consider the following input message, which contains invoice information for two customers:

```
<Data>
  <Invoice>
    <CustomerNumber>1234</CustomerNumber>
    <Item>
      <PartNumber>1</PartNumber>
      <Quantity>9876</Quantity>
    </Item>
    <Item>
      <PartNumber>2</PartNumber>
      <Quantity>8765</Quantity>
    </Item>
  </Invoice>
  <Invoice>
    <CustomerNumber>2345</CustomerNumber>
    <Item>
      <PartNumber>2</PartNumber>
      <Quantity>7654</Quantity>
    </Item>
    <Item>
      <PartNumber>1</PartNumber>
      <Quantity>6543</Quantity>
    </Item>
  </Invoice>
</Data>
```

Consider the following database tables Prices and Addresses and their contents:

PARTNO	PRICE
1	+2.50000E+001
2	+6.50000E+00

PARTNO	STREET	CITY	COUNTRY
1234	22 Railway Cuttings	East Cheam	England
2345	The Warren	Watership Down	England

If you code the following ESQL transform:

```

-- Create a valid output message
SET OutputRoot.MQMD = InputRoot.MQMD;

-- Select suitable invoices
SET OutputRoot.XML.Data.Statement[] =
  (SELECT I.CustomerNumber           AS Customer.Number,
        A.Street                    AS Customer.Street,
        A.City                      AS Customer.Town,
        A.Country                   AS Customer.Country,

-- Select suitable items
        (SELECT II.PartNumber AS PartNumber,
              II.Quantity  AS Quantity,
              PI.Price     AS Price
        FROM Database.db2admin.Prices AS PI,
              I.Item[]      AS II
        WHERE II.PartNumber = PI.PartNo ) AS Purchases.Item[]

FROM Database.db2admin.Addresses AS A,
      InputRoot.XML.Data.Invoice[] AS I

WHERE I.CustomerNumber = A.PartNo
);

```

the following output message is generated. The input message is augmented with the price and address information from the database table:


```

<Data>
  <Statement>
    <Customer>
      <Number>1234</Number>
      <Street>22 Railway Cuttings</Street>
      <Town>East Cheam</Town>
      <Country>England</Country>
    </Customer>
    <Purchases>
      <Item>
        <PartNumber>1</PartNumber>
        <Quantity>9876</Quantity>
        <Price>2.5E+1</Price>
      </Item>
      <Item>
        <PartNumber>2</PartNumber>
        <Quantity>8765</Quantity>
        <Price>6.5E+1</Price>
      </Item>
    </Purchases>
  </Statement>
  <Statement>
    <Customer>
      <Number>2345</Number>
      <Street>The Warren</Street>
      <Town>Watership Down</Town>
      <Country>England</Country>
    </Customer>
    <Purchases>
      <Item>
        <PartNumber>1</PartNumber>
        <Quantity>6543</Quantity>
        <Price>2.5E+1</Price></Item>
      <Item>
        <PartNumber>2</PartNumber>
        <Quantity>7654</Quantity>
        <Price>6.5E+1</Price>
      </Item>
    </Purchases>
  </Statement>
</Data>

```

You can nest the database SELECT within the message SELECT statement. In most cases this is not as efficient as the previous example, but you might find that it is better if the messages are small and the database tables are large.

```

-- Create a valid output message
SET OutputRoot.MQMD = InputRoot.MQMD;

-- Select suitable invoices
SET OutputRoot.XML.Data.Statement[] =
  (SELECT I.CustomerNumber           AS Customer.Number,

    -- Look up the address
    THE ( SELECT
          A.Street,
          A.City   AS Town,
          A.Country
        FROM Database.db2admin.Addresses AS A
        WHERE A.PartNo = I.CustomerNumber
      )
      AS Customer,

    -- Select suitable items
    (SELECT
      II.PartNumber AS PartNumber,
      II.Quantity  AS Quantity,

      -- Look up the price
      THE (SELECT ITEM P.Price
          FROM Database.db2admin.Prices AS P
          WHERE P.PartNo = II.PartNumber
        )
        AS Price

      FROM I.Item[] AS II           ) AS Purchases.Item[]

    FROM InputRoot.XML.Data.Invoice[] AS I
  );

```

Working with XML messages and bit streams:

This topic helps you to use the following ESQL code:

- “The ASBITSTREAM function”
- “The CREATE statement with a PARSE clause” on page 267

The ASBITSTREAM function:

If you code the ASBITSTREAM function with the parser mode option set to RootBitStream to parse a message tree to a bit stream, the result is an XML document that is built from the children of the target element in the normal way. This algorithm is identical to that used to generate the normal output bit stream. Because the target element is not included in the output bit stream, you must ensure that the children of the element follow the constraints for an XML document. One constraint is that there must be only one body element in the message. You can use a well-formed bit stream obtained in this way to recreate the original tree using a CREATE statement with a PARSE clause.

If you code the ASBITSTREAM function with the parser mode option set to FolderBitStream to parse a message tree to a bit stream, the generated bit stream is an XML document built from the target element and its children. Any DocTypeDecl or XmlDecl elements are ignored, and the target element itself is included in the generated bit stream. The advantage of this mode is that the target element becomes the body element of the document, and that body element can have multiple elements nested within it. Use this mode to obtain a bit stream description of arbitrary sub-trees owned by an XML parser. You can use bit streams obtained in this way to recreate the original tree using a CREATE statement with a PARSE clause, and a mode of FolderBitStream.

For further information about ASBITSTREAM and examples of its use, see “ASBITSTREAM function” on page 925.

The CREATE statement with a PARSE clause:

If you code a CREATE statement with a PARSE clause with the parser mode option set to RootBitStream to parse a bit stream to a message tree, the expected bit stream is a normal XML document. A field in the tree is created for each field in the document. This algorithm is identical to that used when parsing a bit stream from an input node. In particular, an element named XML is created as the root element of the tree, and all the content in the message is created as children of that root.

If you code a CREATE statement with a PARSE clause with the parser mode option set to FolderBitStream to parse a bit stream to a message tree, the expected bit stream is a normal XML document. Any content outside the body element (such as an XML declaration or doctype) is discarded. The first element created during the parse corresponds to the body of the XML document, and from there the parse proceeds as normal.

For further information about CREATE and examples of its use, see “CREATE statement” on page 818.

Manipulating messages in the XMLNS domain

This topic provides information specific to dealing with messages that belong to the XMLNS domain, and that are parsed by the generic XML parser. The XMLNS domain is an extension of the XML domain and provides namespace support. Follow the guidance provided for XML messages in “Manipulating messages in the XML domain” on page 239, in conjunction with the information in the topic “Manipulating message body content” on page 171.

The following example shows how to use ESQL to work with namespaces. The example declares namespace constants at the start of the main module so that you can use prefixes in the ESQL statements instead of the full namespace URIs.

The namespace constants affect only the ESQL; they do not control the prefixes generated in the output message. The prefixes in the generated output message are controlled by namespace declarations. You can include namespace declarations in the tree using the XML.NamespaceDecl correlation name. These elements are then used to generate namespace declarations in the output message.

If, when the output message is generated, the namespace with which an element or attribute is qualified has no corresponding namespace declaration, one is automatically generated using prefixes of the form NS n where n is a positive integer.

```

CREATE COMPUTE MODULE xmlns_doc_flow_Compute
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
CALL CopyMessageHeaders();

-- Declaration of namespace constants
DECLARE sp1 NAMESPACE 'http://www.ibm.com/space1';
DECLARE sp2 NAMESPACE 'http://www.ibm.com/space2';
DECLARE sp3 NAMESPACE 'http://www.ibm.com/space3';

-- Namespace declaration to associate prefix 'space1' with the namespace
SET OutputRoot.XMLNS.message.(XML.NamespaceDecl)xmlns:space1 = 'http://www.ibm.com/space1';
SET OutputRoot.XMLNS.message.sp1:data1 = 'Hello!';

-- Default Namespace declaration
SET OutputRoot.XMLNS.message.sp2:data2.(XML.NamespaceDecl)xmlns = 'http://www.ibm.com/space2';
SET OutputRoot.XMLNS.message.sp2:data2.sp2:subData1 = 'Hola!';
SET OutputRoot.XMLNS.message.sp2:data2.sp2:subData2 = 'Guten Tag!';

SET OutputRoot.XMLNS.message.sp3:data3 = 'Bonjour!';

SET OutputRoot.Properties.MessageDomain = 'XMLNS';

RETURN TRUE;
END;

CREATE PROCEDURE CopyMessageHeaders() BEGIN
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
WHILE I < J DO
SET OutputRoot.*[I] = InputRoot.*[I];
SET I = I + 1;
END WHILE;
END;

END MODULE;

```

When this ESQL is processed, the following output message is generated:

```

<message xmlns:space1="http://www.ibm.com/space1">
  <space1:data1>Hello!</space1:data1>
  <data2 xmlns="http://www.ibm.com/space2">
    <subData1>Hola!</subData1>
    <subData2>Guten Tag!</subData2>
  </data2>
  <NS1:data3 xmlns:NS1="http://www.ibm.com/space3">Bonjour!</NS1:data3>
</message>

```

You can also specify that a named XML element (and its descendents, if it is a complex element) is parsed opaquely. That is, a single named element is created in the message tree with a value (encoded in UTF-16) that contains the actual XML bit stream that is contained between the start and end tags of the opaque element. This option can provide performance benefits if the contents of an element are not significant within your message flow.

To specify that an XML element is to be parsed opaquely, use an ESQL CREATE statement with a PARSE clause to parse the XML document. Set the FORMAT qualifier of the PARSE clause to the constant, case-sensitive string 'XMLNS_OPAQUE' and set the TYPE qualifier of the PARSE clause to the name of the XML element which is to be parsed in an opaque manner. The TYPE clause can specify the element name with no namespace (to match any namespace), or with a namespace prefix or full namespace URI (to match a specific namespace).

Consider the following example:

```
DECLARE soap NAMESPACE 'http://schemas.xmlsoap.org/soap/envelope/';

DECLARE BitStream BLOB ASBITSTREAM(InputRoot.XMLNS
                                   ENCODING InputRoot.Properties.Encoding
                                   CCSID InputRoot.Properties.CodedCharSetId);

--No Namespace
CREATE LASTCHILD OF OutputRoot
  DOMAIN('XMLNS')
  PARSE (BitStream
        ENCODING InputRoot.Properties.Encoding
        CCSID InputRoot.Properties.CodedCharSetId
        FORMAT 'XMLNS_OPAQUE'
        TYPE 'Body');

--Namespace Prefix
CREATE LASTCHILD OF OutputRoot
  DOMAIN('XMLNS')
  PARSE (BitStream
        ENCODING InputRoot.Properties.Encoding
        CCSID InputRoot.Properties.CodedCharSetId
        FORMAT 'XMLNS_OPAQUE'
        TYPE 'soap:Body');

--Namespace URI
CREATE LASTCHILD OF OutputRoot
  DOMAIN('XMLNS')
  PARSE (BitStream
        ENCODING InputRoot.Properties.Encoding
        CCSID InputRoot.Properties.CodedCharSetId
        FORMAT 'XMLNS_OPAQUE'
        TYPE '{http://schemas.xmlsoap.org/soap/envelope/}Body');
```

Opaque parsing of XML elements is only available in the XMLNS domain; and the control over how this is specified is subject to change in later releases.

For further information about CREATE and examples of its use, see the “CREATE statement” on page 818.

Manipulating messages using the XMLNSC parser

The XMLNSC domain is an extension of the XMLNS domain, which in turn, was an extension of the original XML domain.

The intent with the XMLNS domain was to add namespace support and, for compatibility reasons, a new domain was created so that existing applications would not be affected. The intent with the new XMLNSC domain is to build a more compact tree and, therefore, use less memory when handling large messages. Again, for compatibility reasons, a new domain has been added so that existing applications are not affected.

Message tree structure

The XMLNSC parser obtains its more compact tree by using a single name-value element to represent tagged text, rather than the separate name and value elements used by the XML and XMLNS parsers. Consider the following message:

```
<Folder1>
  <Folder2 Attribute1='AttributeValue1'>
    <Field1><Value1></Field1>
    <Field2 Attribute2='AttributeValue2'><Value2></Field2>
  </Folder2>
</Folder1>
```

In the XMLNSC domain, this is represented by two name elements (Folder1 and Folder2) and four name-value elements which are Attribute1, Field1, Field2, and Attribute2.

The XML and XMLNS domains differ in that the two fields are each represented by a name element with a child value element. This might seem to be a small difference, but messages often have many such leaf fields; for example:

```
<Folder1>
  <Folder2>
    <Field1><Value1></Field1>
    <Field2><Value2></Field2>
    ...
    <Field100><Value100></Field100>
  </Folder2>
</Folder1>
```

In this case, the XMLNSC parser represents the message by two name and 100 name-value elements, whereas the XML and XMLNS parsers would use 102 name elements and 100 value elements, plus a further 103 value elements to represent the white-space implicit in formatted messages.

Attributes and tagged text

As both attributes and tagged text are represented by name-value elements, they are distinguished by the use of the element types. If you do not specify a type, tagged text is assumed. Therefore, the first example message above might be produced by the SQL statements:

```
SET Origin.Folder1.Folder2.(XMLNSC.Attribute)Attribute1 =
  'AttributeValue1';
SET Origin.Folder1.Folder2.Field1 = 'Value1';
SET Origin.Folder1.Folder2.(XMLNSC.Attribute)Attribute2 =
  'AttributeValue2';
SET Origin.Folder1.Folder2.Field2 = 'Value2';
```

Although the preceding SQL looks almost identical to that which would be used with the XML parser, note particularly that the type constants being used are ones that belong to the XMLNSC parser. The use of constants that belong to other parsers, for example XML, leads to unexpected results because similarly named constants, for example XML.Attribute, have different values.

Handling mixed text

By default, mixed text is simply discarded on the grounds that, if present, it is simply formatting and has no meaning.

However, a mode is provided in which, when parsing, any text that occurs other than between an opening tag and a closing tag (that is, open->open, close->close, and close->open) is represented by a single Value element. The value element types support PCDATA, CDATA, and hybrid which is a mixture of the preceding two.

There is still no special syntax element behavior regarding the getting and setting of values. Value elements can only be accessed from the SQL by explicitly addressing them. The following extra constants are provided for this purpose:

```
XMLNSC.Value
XMLNSC.PCDataValue
XMLNSC.CDataValue
XMLNSC.HybridValue
```

The mode is controlled by new message option values. For this purpose, the following constants are provided:

```
XMLNSC.MixedContentRetainNone = 0x0000000000000000
XMLNSC.MixedContentRetainAll  = 0x0001000000000000
```

These constants can be used in the Option clauses of both the SQL “CREATE statement” on page 818 (PARSE section) and the “ASBITSTREAM function” on page 925. For example:

```
DECLARE X BLOB ASBITSTREAM(InputRoot.XMLNSC.Data OPTIONS
XMLNSC.MixedContentRetainAll);
...
CREATE LASTCHILD OF outputRoot PARSE(X OPTIONS
XMLNSC.MixedContentRetainNone);
```

Handling comments

By default, comments are also simply discarded on the grounds that, if present, they are simply auxiliary information with no meaning.

However, a mode is provided in which, when parsing, any comments that occur in the document (other than in the document description itself) are represented by a name-value element with the name Comment. The following extra comment is provided for this purpose.

```
XMLNSC.Comment
```

The mode is controlled by new message option values. The following constants are provided for this purpose:

```
XMLNSC.CommentsRetainNone = 0x0000000000000000
XMLNSC.CommentsRetainAll  = 0x0002000000000000
```

For example:

```
DECLARE X BLOB ASBITSTREAM(InputRoot.XMLNSC.Data OPTIONS
XMLNSC.CommentsRetainAll);
...
CREATE LASTCHILD OF outputRoot PARSE(X OPTIONS XMLNSC.CommentsRetainNone);
```

Handling processing instructions

By default, processing instructions are also simply discarded on the grounds that, if present, they are simply auxiliary information with no meaning.

However, a mode is provided in which when parsing, any processing instructions that occur in the document (other than in the document description itself) are represented by a name-value element with the appropriate name and value. The following extra constant is provided for this purpose:

```
XMLNSC.ProcessingInstruction
```

The mode is controlled by new message option values. The following constants are provided for this purpose:

```
XMLNSC.ProcessingInstructionsRetainNone = 0x0000000000000000
XMLNSC.ProcessingInstructionsRetainAll  = 0x0004000000000000
```

For example:

```

DECLARE X BLOB ASBITSTREAM(InputRoot.XMLNSC.Data
OPTIONS XMLNSC.ProcessingInstructionsRetainAll);
...
CREATE LASTCHILD OF outputRoot PARSE(X OPTIONS
XMLNSC.ProcessingInstructionsRetainNone);

```

Migrating an existing flow

The fact that a new domain has been introduced means that, when using the XMLNSC parser, you must re-code your ESQL to use XMLNSC in your paths. Consider the following examples:

```

SET OutputRoot.XML.Person.Salary =
    CAST(InputRoot.XML.Person.Salary AS INTEGER) * 3;
SET OutputRoot.XMLNS.Person.Salary =
    CAST(InputRoot.XMLNS.Person.Salary AS INTEGER) * 3;
SET OutputRoot.XMLNSC.Person.Salary =
    CAST(InputRoot.XMLNSC.Person.Salary AS INTEGER) * 3;

```

In each case the XML bit-stream expected at the input queue and written to the output queue is of the form:

```
<Person><Salary>42</Salary></Person>
```

The three cases differ in that they are using different parsers to own these elements. Therefore, a different domain name is expected in the MQRFH2 header of the incoming message and a different domain name is written in the MQRFH2 header of the outgoing message.

To protect external applications from these changes, the Use XMLNSC Compact Parser for XMLNS Domain property can be specified on the flow's input node, and on the compute node containing these statements.

The first example causes the XMLNSC parser to be used to parse the body of the message when the MQRFH2 header in the incoming message specifies the XMLNS domain; that on the compute node causes the outgoing MQRFH2 to specify the XMLNS instead of XMLNSC parser, so allowing the input and output messages to remain unchanged.

If the incoming messages do not contain MQRFH2 headers, and the input node's message domain attribute is being used to specify the domain, you can either set it to XMLNSC, or set it to XMLNS and also set the Use XMLNSC Compact Parser for XMLNS Domain property.

If outgoing messages do not contain MQRFH2 headers, the domain does not appear anywhere in the output messages and the setting of the compute node's Use XMLNSC Compact Parser for XMLNS Domain property has no effect

Constructing XML headers

The following ESQL is valid in the XML domain:

```
SET OutputRoot.XML.(XML.XmlDecl)*.(XML.Version)* = '1.0';
```

To migrate to XMLNS, simply changing the root is enough to make this work:

```
SET OutputRoot.XMLNS.(XML.XmlDecl)*.(XML.Version)* = '1.0';
```

Note that although the XMLNS parser is being used, the element type constants are those belonging to the XML parser. This works because the type values used by the

XML and XMLNS parsers are the same. For the XMLNSC parser, however, the type values are different and, therefore, you *must* always use its own type constants.

In the XMLNSC domain there is no special type for the XML version; it is simply treated as an attribute of the XML declaration. The equivalent syntax for the above example is:

```
SET OutputRoot.XMLNSC.(XMLNSC.XmlDeclaration)*.(XMLNSC.Attribute)Version = '1.0';
```

Copying message trees

When copying trees, the broker regards XML and XMLNSC as unlike parsers, which means that all attributes in the source tree get mapped to elements in the target tree. This situation arises only if you are using both parsers in the same flow - one for input and one for output; you are therefore recommended to use the compact parser for both flows.

If different parsers *must* be used for the input flow and output flow, you might need to explicitly specify the types of elements in the paths or use the “FIELDVALUE function” on page 933 to ensure a copy of scalar values rather than of sub-trees.

Follow the guidance provided for XML messages in “Manipulating messages in the XML domain” on page 239, in conjunction with the information in the topic “Manipulating message body content” on page 171.

Accessing syntax elements in the XMLNSC domain using correlation names

The following table provides the correlation names for each XML syntax element. When working in the XMLNSC domain, use these names to refer to the elements in input messages, and to set elements, attributes, and values in output messages.

Table 1. Correlation names for XML syntax elements

Syntax element	Correlation name	Constant value
Folder	XMLNSC.Folder	0x01000000
Document type ¹	XMLNSC.DocumentType	0x01000300
XML declaration ²	XMLNSC.XmlDeclaration	0x01000400
Field or Attr Value	XMLNSC.Value	0x02000000
PCData value	XMLNSC.PCDataValue	0x02000000
CData value	XMLNSC.CDataValue	0x02000001
Hybrid value	XMLNSC.HybridValue	0x02000002
Entity Reference	XMLNSC.EntityReference	0x02000100
Field	XMLNSC.Field	0x03000000
PCData	XMLNSC.PCDataField	0x03000000
CData	XMLNSC.CDataField	0x03000001
Hybrid	XMLNSC.HybridField	0x03000002
Attribute	XMLNSC.Attribute	0x03000100

Table 1. Correlation names for XML syntax elements (continued)

Syntax element	Correlation name	Constant value
Single quote	XMLNSC.SingleAttribute	0x03000101
Double quote	XMLNSC.DoubleAttribute	0x03000100
Namespace declaration	XMLNSC.NamespaceDecl	0x03000102
Single quote	XMLNSC.SingleNamespaceDecl	0x03000103
Double quote	XMLNSC.DoubleNamespaceDecl	0x03000102
Bitstream data	XMLNSC.BitStream	0x03000200
Entity definition ¹	XMLNSC.EntityDefinition	0x03000300
Single quote	XMLNSC.SingleEntityDefinition	0x03000301
Double quote	XMLNSC.DoubleEntityDefinition	0x03000300
Comment	XMLNSC.Comment	0x03000400
Processing instruction	XMLNSC.ProcessingInstruction	0x03000401

Notes:

- Document Type is only used for entity definitions. For example:

```
SET OutputRoot.XMLNSC.(XMLNSC.DocumentType)BodyDocument
    .(XMLNSC.EntityDefinition)TestDef =
    'Compact Tree Parser XML Test Module Version 1.0';
```
- Note that the XML declaration is a special folder type that contains child elements for version, and so on. For example:

```
-- Create the XML declaration
SET OutputRoot.XMLNSC.(XMLNSC.XmlDeclaration)*.Version = 1.0;
SET OutputRoot.XMLNSC.(XMLNSC.XmlDeclaration)*.Encoding = 'UTF8';
SET OutputRoot.XMLNSC.(XMLNSC.XmlDeclaration)*.Standalone = 'yes';
```

XMLNSC parser modes

By default, the XMLNSC parser discards document elements that typically carry no business meaning. However, parser modes are available to force retention of these elements. You can configure these modes on the properties of the node that specifies the message is to be parsed in the XMLNSC domain.

The valid parser modes for the XMLNSC parser are:

```
XMLNSC.MixedContentRetainNone
XMLNSC.MixedContentRetainAll
XMLNSC.CommentsRetainNone
XMLNSC.CommentsRetainAll
XMLNSC.ProcessingInstructionsRetainNone
XMLNSC.ProcessingInstructionsRetainAll
```

The following example uses the XMLNSC.ProcessingInstructionsRetainAll and XMLNSC.ProcessingInstructionsRetainNone modes to retain document processing instructions while parsing:

```

DECLARE X BLOB ASBITSTREAM(InputRoot.XMLNSC.Data OPTIONS XMLNSC
                          .ProcessingInstructionsRetainAll);
...
CREATE LASTCHILD OF outputRoot PARSE(X OPTIONS XMLNSC
                          .ProcessingInstructionsRetainNone);

```

Manipulating messages in the JMS domains

This topic provides information specific to dealing with messages that belong to the JMS domains, and that are parsed by the generic XML parser. Because they are processed by the same parser, you can follow the guidance provided for XML messages in “Manipulating messages in the XML domain” on page 239, in conjunction with the information in “Manipulating message body content” on page 171.

You can create messages with JMS types `jms_map` and `jms_stream` messages: no other categories of JMS messages are supported. For further information about using JMS messages with WebSphere Message Broker, see the *WebSphere MQ Using Java* book.

Manipulating messages in the IDoc domain

A valid IDoc message flows out of SAP and is sent to the MQSeries link for R/3.

When this IDoc has been successfully committed to the outbound WebSphere MQ queue, the input node of the message flow reads it from that queue and generates the syntax element tree.

The Compute node manipulates this syntax element tree and, when it has finished, passes the output message to subsequent nodes in the message flow. When the message reaches the output node, the IDoc parser is called to rebuild the bit stream from the tree.

The message flow must create an output message in a similar format to the input message.

See “Field names of the IDoc parser structures” on page 740 for the field names in the DC and DD recognized by the IDoc parser

Use the following ESQL as an example from a Compute node:

```

SET OutputRoot = InputRoot;
SET OutputRoot.IDOC.DC[1].tabnam = 'EDI_DC40 ';
SET OutputRoot.IDOC.DD[2].sdatatag.MRM.maktx = 'Buzzing all day';

```

The first line copies the incoming IDoc to the outgoing IDoc.

The second line sets the *tabname* of the first DC.

The third line uses the second DD segment, which in this example is of type E2MAKTM001, and sets the *maktx* field.

Manipulating messages in the MIME domain

This topic explains how to deal with messages that belong to the MIME domain, and that are parsed by the MIME parser. Use this information in conjunction with the information in “Manipulating message body content” on page 171.

A MIME message does not have to be received over a particular transport. For example a message can be received over HTTP using an HTTPInput node, or over WebSphere MQ using an MQInput node. The MIME parser is used to process a message if one of the following conditions applies:

- The message domain is set to MIME in the input node properties.
- You are using WebSphere MQ and the MQRFH2 header has a message domain of MIME.

The logical tree can be manipulated using ESQL before the message is passed on to other nodes in the message flow. A message flow can also create a MIME domain tree using ESQL. When a MIME domain message reaches an output node, the MIME parser is called to rebuild the bit stream from the logical tree.

The following examples show how to manipulate MIME messages:

- “Creating a new MIME tree”
- “Modifying an existing MIME tree” on page 277
- “Managing Content-Type” on page 277

Creating a new MIME tree

A message flow often receives, modifies and returns a MIME message. In this case you can work with the valid MIME tree created when the input message is parsed. If a message flow receives input from another domain, such as XML, and returns a MIME message you need to create a valid MIME tree. Use the following ESQL in a Compute node to create the top-level structure for a single-part MIME tree:

```
CREATE FIELD OutputRoot.MIME TYPE Name;  
DECLARE M REFERENCE TO OutputRoot.MIME;  
CREATE LASTCHILD OF M TYPE Name NAME 'Data';
```

The message flow also needs to ensure that the MIME Content-Type is set correctly, as explained in “Managing Content-Type” on page 277. The flow then needs to add the message data into the MIME tree. The following ESQL gives examples of how this can be done. Note that in each case Data is created with the domain BLOB.

- A bit stream from another part of the tree is used. The following example shows how a bit stream could be created from an XML message that the message flow received. The flow then invokes the BLOB parser to store the data under the Data element.

```
DECLARE partData BLOB ASBITSTREAM(InputRoot.XML);  
CREATE LASTCHILD OF M.Data DOMAIN('BLOB') PARSE(partData);
```

- Instead of parsing the bit stream, create the new structure then attach the data to it. The following ESQL is an example of how to do this:

```
DECLARE partData BLOB ASBITSTREAM(InputRoot.XML);  
CREATE LASTCHILD OF M.Data DOMAIN('BLOB') NAME 'BLOB';  
CREATE LASTCHILD OF M.Data.BLOB NAME 'BLOB' VALUE partData;
```

Both of these approaches create the same tree structure. The first approach is recommended because explicit knowledge of the tree structure that the BLOB parser requires is not built into the flow.

More commonly, the Compute node needs to build a tree for a multipart MIME document. The following ESQL is an example of how you can do this, including setting the top-level Content-Type via the ContentType property:

```

DECLARE part1Data BLOB ASBITSTREAM(InputRoot.XML.part1);
DECLARE part2Data BLOB ASBITSTREAM(InputRoot.XML.part2);

SET OutputRoot.Properties.ContentType = 'multipart/related; boundary=myBoundary';

CREATE FIELD OutputRoot.MIME TYPE Name;
DECLARE M REFERENCE TO OutputRoot.MIME;
CREATE LASTCHILD OF M TYPE Name NAME 'Parts';
CREATE LASTCHILD OF M.Parts TYPE Name NAME 'Part';
DECLARE P1 REFERENCE TO M.Parts.Part[1];
CREATE FIELD P1."Content-Type" TYPE NameValue VALUE 'text/plain';
CREATE FIELD P1."Content-Id" TYPE NameValue VALUE 'part one';
CREATE LASTCHILD OF P1 TYPE Name NAME 'Data';
CREATE LASTCHILD OF P1.Data DOMAIN('BLOB') PARSE(part1Data);

CREATE LASTCHILD OF M.Parts TYPE Name NAME 'Part';
DECLARE P2 REFERENCE TO M.Parts.Part[2];
CREATE FIELD P2."Content-Type" TYPE NameValue VALUE 'text/plain';
CREATE FIELD P2."Content-Id" TYPE NameValue VALUE 'part two';
CREATE LASTCHILD OF P2 TYPE Name NAME 'Data';
CREATE LASTCHILD OF P2.Data DOMAIN('BLOB') PARSE(part2Data);

```

Modifying an existing MIME tree

This example ESQL adds a new MIME part to an existing multipart MIME message. If the message is not multipart it is not modified:

```

SET OutputRoot = InputRoot;

-- Check to see if the MIME message is multipart or not.
IF LOWER(InputProperties.ContentType) LIKE 'multipart/%'
THEN
  CREATE LASTCHILD OF OutputRoot.MIME.Parts NAME 'Part';

  DECLARE P REFERENCE TO OutputRoot.MIME.Parts.[<];
  CREATE FIELD P."Content-Type" TYPE NameValue VALUE 'text/xml';
  CREATE FIELD P."Content-ID" TYPE NameValue VALUE 'new part';
  CREATE LASTCHILD OF P TYPE Name NAME 'Data';

  -- This is an artificial way of creating some BLOB data.
  DECLARE newBlob BLOB '4f6e652074776f2074687265650d0a';
  CREATE LASTCHILD OF P.Data DOMAIN('BLOB') PARSE(newBlob);
END IF;

```

Managing Content-Type

When you create a new MIME message tree, or when you modify the value of the MIME boundary string, you must make sure that the MIME Content-Type header is set correctly. Set the ContentType value in the broker Properties subtree to do this. The following example shows the ContentType value being set for a MIME part with simple content:

```
SET OutputRoot.Properties.ContentType = 'text/plain';
```

Do not set the Content-Type value directly in the MIME tree or HTTP trees. This can lead to the value being ignored or used inconsistently.

Manipulating messages in the BLOB domain

This topic provides information specific to dealing with messages that belong to the BLOB domain, and that are parsed by the BLOB parser.

You cannot manipulate the contents of a BLOB message, because it has no predefined structure. However, you can refer to its contents using its known position within the bit stream, and process the message with a minimum of knowledge about its contents.

The BLOB message body parser does not create a tree structure in the same way that other message body parsers do. It has a root element BLOB, that has a child element, also called BLOB, that contains the data.

You can refer to message content using substrings if you know the location of a particular piece of information within the BLOB data. For example, the following expression identifies the tenth byte of the message body:

```
InputBody.BLOB.BLOB[10]
```

The following expression references 10 bytes of the message data starting at offset 10:

```
SUBSTRING(InputBody.BLOB.BLOB from 10 for 10)
```

Example of BLOB message manipulation:

This example shows how to manipulate a variable length BLOB message. The example assumes that you have configured a message flow that receives a variable length BLOB message, parses some of the fields by invoking the MRM parser, and routes the output message to the correct output queue based on the information parsed.

The input message is in BLOB format and is assumed to contain embedded NULLs ('x00'), so it cannot be defined as null terminated.

This example shows the ESQL needed to:

- Calculate the BLOB message length
- Convert it to hexadecimal format
- Add it to the beginning of the BLOB message

By doing this, you can define the message model with an integer length field followed by the BLOB message.

This example also shows how to convert the BLOB message to CWF, process the message, and strip off the added length field.

In this example the input record has the following format:

- Version Number: string, 11 characters
- Store Number: string, 10 characters
 - This field is used as an integer to route the message to different queues depending on customer-defined criteria.
- Store Data: variable length binary data

Define a new message:

Define a new message BLOB_Example that includes the following elements and types:

- B_LEN, xsd:integer
- VERSION_NUM, xsd:string, Length 11

- STORE_NUM, xsd:string, Length 10
- BIN_BLOB, xsd:binary, Length Value B_LEN

Create a message flow:

This section describes the characteristics of the message flow. If you want to implement this example flow, you must complete the message flow definition (for example, by creating the three subflows to replace the output nodes used here to handle false, unknown, and failure cases) and provide any required support for its deployment and execution (for example, creating the inbound and any outbound queues on the queue manager for the broker to which you deploy the flow).

1. Create the subflow LESS_THAN. This task is described in “Create LESS_THAN subflow” on page 281.
2. Create a new message flow. Add nodes to the message flow editor view: an MQInput node, a Compute node, a ResetContentDescriptor node, a Filter node, three MQOutput nodes, and the LESS_THAN subflow.
3. Change the name of the MQInput node to INQUEUE and set its *Queue Name* property to INQUEUE.
4. Connect the output terminal to the Compute node.
5. Change the Compute node name from its default value to Add_length. Configure the Compute node to calculate the length of the BIN_BLOB and add it to the beginning of the BLOB_Example message in field B_LEN:
 - a. Right-click the node and click **Open ESQL**.
 - b. Code the following ESQL in the module for this node:

```

-- Declare local variables
DECLARE I      INTEGER 1;
DECLARE J      INTEGER CARDINALITY(InputRoot.*[]);
DECLARE MSGLEN CHARACTER;
DECLARE NUMBER INTEGER;
DECLARE RESULT INTEGER;
DECLARE REM    INTEGER;

-- Copy message headers
WHILE I < J DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I = I + 1;
END WHILE;
--
-- Set MSGLEN to non NULL to avoid errors when concatenating the first time --
SET MSGLEN = 'X';
--
-- Get the length of the BLOB and subtract the length of VERSION_NUM and STORE_NUM (11+10)
SET NUMBER = LENGTH("InputRoot"."BLOB"."BLOB")-21;
--
-- Convert NUMBER to hexadecimal. The remainder of dividing by 16 is calculated recursively. --
WHILE NUMBER > 15 DO
  SET RESULT = NUMBER/16;
  SET REM    = NUMBER - RESULT*16;
  SET MSGLEN =
  CASE
    WHEN REM < 10 THEN CAST(REM AS CHARACTER) || MSGLEN
    WHEN REM = 10 THEN 'A' || MSGLEN
    WHEN REM = 11 THEN 'B' || MSGLEN
    WHEN REM = 12 THEN 'C' || MSGLEN
    WHEN REM = 13 THEN 'D' || MSGLEN
    WHEN REM = 14 THEN 'E' || MSGLEN
    ELSE 'F' || MSGLEN
  END;
  SET NUMBER = RESULT;
END WHILE;
SET REM = NUMBER;
SET MSGLEN =
CASE
  WHEN REM < 10 THEN CAST(REM AS CHARACTER) || MSGLEN
  WHEN REM = 10 THEN 'A' || MSGLEN
  WHEN REM = 11 THEN 'B' || MSGLEN
  WHEN REM = 12 THEN 'C' || MSGLEN
  WHEN REM = 13 THEN 'D' || MSGLEN
  WHEN REM = 14 THEN 'E' || MSGLEN
  ELSE 'F' || MSGLEN
END;
--
-- Add leading '0's up to a length of 9 to be able to cast as BLOB.
-- Remember it started with MSGLEN set to X (length 1)
WHILE LENGTH(MSGLEN) < 9 DO
  SET MSGLEN = '0' || MSGLEN; END WHILE;
--
-- Change to appropriate endian (PLATFORM DEPENDENT)
-- If no endian swapping needed then remember to get rid of the last character as below --
SET MSGLEN = SUBSTRING(MSGLEN FROM 1 FOR 8);
--
SET MSGLEN = SUBSTRING(MSGLEN FROM 7 FOR 2) || SUBSTRING(MSGLEN FROM 5 FOR 2) ||
  SUBSTRING(MSGLEN FROM 3 FOR 2) || SUBSTRING(MSGLEN FROM 1 FOR 2);
SET "OutputRoot"."BLOB"."BLOB" = CAST(MSGLEN AS BLOB) || "InputRoot"."BLOB"."BLOB";

```

6. Connect the out terminal of the Compute node to the ResetContentDescriptor node.
7. Change the ResetContentDescriptor node name to ResetContent_2_MRM. Configure the node as follows:

- a. Set *Message Domain* to MRM.
 - b. Select the *Reset Message Domain* check box.
 - c. Set *Message Set* to the identifier of the message set in which you defined the BLOB_Example message.
 - d. Select the *Reset Message Set* check box.
 - e. Set *Message Type* to BLOB_Example.
 - f. Select the *Reset Message Type* check box.
 - g. Set to the name of the CWF physical format that you have defined (for example, the default value CWF1).
 - h. Select the *Reset Message Format* check box.
8. Connect the out terminal of the ResetContentDescriptor node to the Filter node.
 9. Change the name of the Filter node to Route_2_QUEUE. Configure the node as follows:
 - a. Right-click the node and click **Open ESQL**.
 - b. Code the following ESQL statement in the ESQL module for this node:

```
CAST("Body"."e_STORE_NUM" AS INTEGER) < 151
```

This statement is based on the arbitrary assumption that an incoming message from a Store Number is less than 151 and is routed to a specific queue. You can code any other suitable test.

10. Connect the Filter output terminals as follows:
 - a. True terminal to a subflow node (see below) named LESS_THAN.
 - b. False terminal to an MQOutput node named GREATER_THAN with *Queue Name* property set to GREATER_THAN.
 - c. Unknown terminal to an MQOutput node named INVALID with *Queue Name* property set to INVALID.
 - d. Failure to an MQOutput node named ERROR with *Queue Name* property set to ERROR.

Create *LESS_THAN* subflow:

This subflow handles a message that has the expected format (the test performed in the Filter node returned true). The successful message is written to the output queue in its original form; the message is converted back to BLOB from MRM and the four bytes that were added (field B_LEN) are removed.

For this subflow:

1. Create a new message flow named LESS_THAN.
2. In the editor view, add an Input node, a ResetContentDescriptor node, a Compute node, and an MQOutput node.
3. Change the name of the Input node to InputTerminal1 and connect its out terminal to the ResetContentDescriptor node.
4. Change the name of the ResetContentDescriptor to ResetContent_2_BLOB and configure the node:
 - a. Set *Message Domain* to BLOB
 - b. Select the *Reset Message Domain* check box.
5. Connect the ResetContentDescriptor node out terminal to the Compute node.

6. Change the name of the Compute node to Remove_length and configure the node:
 - a. Right-click the node and click **Open ESQL**.
 - b. Code the following ESQL in the module for this node:

```
-- Copy message headers
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);

WHILE I < J DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I = I + 1;
END WHILE;
--
-- Remove the 4 bytes length field added previously --
SET "OutputRoot"."BLOB"."BLOB" = SUBSTRING("InputRoot"."BLOB"."BLOB" FROM 5);
```

This Compute node removes the four bytes that were added at the beginning of the BLOB message to support its manipulation.

Note the use of a variable, J, initialized to the value of the cardinality of the existing headers in the message. This is more efficient than calculating the cardinality on each iteration of the loop, which happens if you code the following WHILE statement:

```
WHILE I < CARDINALITY(InputRoot.*[]) DO
```

7. Connect the out terminal of the Compute node to the MQOutput node.
8. Change the name of the MQOutput node to Output_success and configure the node, setting *Queue Manager Name* and *Queue Name*. You might find it helpful to promote these MQOutput node properties so that you can specify them at the message flow level.

Using the CALL statement to invoke a user-written routine

The ESQL CALL statement invokes a routine. A routine is a user-defined function or procedure that has been defined by one of the following:

- A CREATE FUNCTION statement
- A CREATE PROCEDURE statement

Note: As well as standard user-defined functions and procedures, you can also use CALL to invoke built-in (broker-provided) functions and user-defined SQL functions. However, the usual way of invoking these types of function is simply to use their names in expressions.

You can use the CALL statement to invoke a routine that has been implemented in any of the following ways:

- ESQL.
- Java.
- As a stored procedure in a database.
- As a built-in (broker-provided) function. (But see the note above about calling built-in functions.)

For details of the syntax and parameters of the CALL statement, see “CALL statement” on page 813. For an example of the use of CALL, see the examples in “CREATE PROCEDURE statement” on page 837.

Calling an ESQL routine:

A routine is invoked as an ESQL method if the routine's definition specifies a LANGUAGE clause of ESQL or if the routine is a built-in function.

There must be an exact one-to-one matching, between the definition and the CALL, of the data types and directions of each parameter.

An ESQL routine is allowed to return any ESQL data type, excluding List and Row.

Calling a Java routine:

A routine is invoked as a Java method if the routine's definition specifies a LANGUAGE clause of JAVA.

There must be an exact one-to-one matching, between the definition and the CALL, of the data types and directions of each parameter.

If the Java method has a void return type, the INTO clause cannot be used because there is no value to return.

A Java routine can return any data type in the "ESQL-to-Java data-type mapping table" on page 790. Note that this excludes List and Row.

Calling a database stored procedure:

A routine is invoked as a database stored procedure if the routine's definition has a LANGUAGE clause of DATABASE.

When a call is made to a database stored procedure, the broker searches for a definition (created by a CREATE PROCEDURE statement) that matches the procedure's local name. The broker then uses the following sequence to resolve the name by which the procedure is known in the database and the database schema to which it belongs:

1. If the CALL statement specifies an IN clause, the name of the data source, the database schema, or both, is taken from the IN clause.
2. If the name of the data source is not provided by an IN clause on the CALL statement, it is taken from the DATASOURCE attribute of the node.
3. If the database schema is not provided by an IN clause on the CALL statement, but is specified on the EXTERNAL NAME clause of the CREATE PROCEDURE statement, it is taken from the EXTERNAL NAME clause.
4. If no database schema is specified on the EXTERNAL NAME clause of the CREATE PROCEDURE statement, the database's user name is used as the schema name. If a matching procedure is found, the routine is invoked.

The chief use of the CALL statement's IN clause is that it allows the data source, the database schema, or both to be chosen dynamically at run time.

Note: As well as the IN clause of the CALL statement, the EXTERNAL SCHEMA clause too, allows the database schema which contains the stored procedure to be chosen dynamically, but it is not as flexible as the IN clause and is retained only for backward compatibility. Its use in new applications is deprecated.

If the called routine has any DYNAMIC RESULT SETS specified in its definition, the number of expressions in the CALL statement's *ParameterList* must match the

number of actual parameters to the routine, plus the number of DYNAMIC RESULT SETS. For example, if the routine has three parameters and two DYNAMIC RESULT SETS, the CALL statement must pass five parameters to the called routine. The parameters passed for the two DYNAMIC RESULT SETS must be list parameters; that is, they must be field references qualified with array brackets []; for example, Environment.ResultSet1[].

A database stored procedure is allowed to return any ESQL data type, excluding Interval, List, and Row.

Accessing broker properties from ESQL

It can be useful, during the runtime of your code, to have real-time access to details of a specific node, flow, or broker. For an overview of broker properties, see “Broker properties” on page 49.

You can use broker properties on the right side of regular SET statements. For example:

```
DECLARE mybroker CHARACTER;  
SET mybroker = BrokerName;
```

where BrokerName is the broker property that contains the broker’s name. However, you cannot use broker properties on the left-hand side of SET statements. This is because, at runtime, broker properties are constants: they cannot be assigned to, and so their values cannot be changed by SET statements. If a program tries to change the value of a broker property, the error message Cannot assign to a symbolic constant is issued.

Broker properties:

- Are grouped by broker, execution group, flow, and node.
- Are case sensitive. Their names always start with an uppercase letter.
- Return NULL if they do not contain a value.

If your ESQL code already contains a variable with the same name as one of the broker properties, your variable takes precedence; that is, your variable masks the broker property. To access the broker property, use the form SQL.<broker_property_name>. For example: SQL.BrokerName.

“Broker properties accessible from ESQL and Java” on page 983 shows the broker, flow, and node properties that are accessible from ESQL and indicates which properties are also accessible from Java.

Configuring a message flow at deployment time using UDPs

User-defined properties (UDPs) give you the opportunity to configure message flows at deployment time, without modifying program code.

A UDP is a user-defined constant that you can use in your ESQL or Java programs. You can give the UDP an initial value when you declare it in your program, or when you use the Message Flow editor to create or modify a message flow.

In ESQL, you can define UDPs at the module or schema level.

For an overview of user-defined properties, see “User-defined properties” on page 50.

After a UDP has been defined by the Message Flow editor, you can modify its value before you deploy:

1. From the workbench, switch to the Broker Administration perspective.
2. Double click your bar file in the Broker Administration Navigator view. The contents of the bar file are shown in the Content editor.
3. Select the Configure tab at the bottom of the Content editor pane. This shows the names of your message flows; these can be expanded to show the individual nodes that are contained in the flow.
4. Click on a message flow name. The UDPs that are defined in that message flow are displayed with their values.
5. If the value of the UDP is unsuitable for your current environment or task, change it to the value that you want. The value of the UDP is set at the flow level and is the same for all eligible nodes that are contained in the flow. If a subflow includes a UDP that has the same name as a UDP in the main flow, the value of the UDP in the subflow is not changed.

Now you are ready to deploy the message flow. See “Deploying a broker archive file” on page 434.

Developing Java

When you use the built-in JavaCompute node, customize it to determine the exact processing that it provides. To do this, create a Java class file for each node in which you code Java functions, to tailor the behavior of the node. Java files are managed through the Java perspective.

You can add any valid Java code to a JavaCompute node, making full use of the existing Java user-defined node API to process an incoming message. You can use the Java editing facilities of the Eclipse platform to develop your Java code. These facilities include:

- code completion
- integrated Javadoc documentation
- automatic compilation

The Java user-defined node API includes some extra methods that simplify tasks that involve message routing and transformation. These tasks include accessing named elements in a message tree, setting their values, and creating elements, without the need to navigate the tree explicitly.

Use the Debug perspective to debug a message flow that contains a JavaCompute node. When control passes to a JavaCompute node during debugging, the perspective opens the Java debugger, allowing you to step through the Java class code for the node.

This section provides the following information on developing Java:

- “Managing Java Files” on page 286
- “Writing Java” on page 288
- Java user-defined node API

Managing Java Files

The Java code that you provide to modify or customize the behavior of a JavaCompute node is stored in a Java project. WebSphere Message Broker uses the Eclipse Java perspective for developing and administering Java files.

This section contains topics that describe how to manage these files:

- “Creating Java code for a JavaCompute node”
- “Opening an existing Java file” on page 287
- “Saving a Java file” on page 287
- “Adding Java code dependencies” on page 287
- “Deploying JavaCompute node code” on page 288

Creating Java code for a JavaCompute node

Before you start

To complete this task, you must have added a “JavaCompute node” on page 542 to your message flow.

To associate code with a Java compute node, use one of the following methods:

- Use the JavaCompute node wizard to create template code.
- Associate a JavaCompute node with an existing Java class that the wizard previously generated.

To generate template code perform the following steps:

1. Right-click the node and click **Open Java**.
2. Step through the JavaCompute node wizard until you reach the Java Compute Node Class Template page. On this page choose one of the following options:
 - For a filter node template code, choose **Filtering Message Class**.
 - To change an incoming message, choose **Modifying Message Class**.
 - To create a new message, choose **Creating Message Class**.

You have now created template code for your JavaCompute node.

Alternatively, you can associate a JavaCompute node with an existing Java class that the wizard previously generated. This allows you to share the same Java code between multiple nodes. To associate a JavaCompute nodes with an existing Java class perform the following steps:

1. Right-click the JavaCompute node and click **Properties**.
2. Enter the name of the Java class in the **Java Class** field.
3. Click **OK**. You have now associated your JavaCompute node with an existing Java class.

You can now perform the following tasks:

- “Opening an existing Java file” on page 287
- “Saving a Java file” on page 287
- “Adding Java code dependencies” on page 287

Opening an existing Java file

You can add to and modify Java code that you have created in a Java project.

Before you start

Before you start this task, complete the following tasks:

- Add a “JavaCompute node” on page 542 to your message flow.
- “Creating Java code for a JavaCompute node” on page 286

To open an existing Java file:

1. Switch to the Java perspective.
2. In the Package Explorer view, double-click the Java file that you want to open. The file is opened in the editor view.
3. Work with the contents of the file to make your changes.

You can also open a Java file when you have a message flow open in the editor view. Select the JavaCompute node, right-click and then select **Open Java** to open the file.

Next:

You can now perform the following tasks:

- “Saving a Java file”
- “Adding Java code dependencies”

Saving a Java file

When you edit your Java files, save them to preserve the additions and modifications that you have made.

Before you start

To complete this task, you must have completed the following tasks:

- Add a “JavaCompute node” on page 542 to your message flow.
- “Creating Java code for a JavaCompute node” on page 286

To save a Java file:

1. Switch to the Java perspective.
2. Create a new Java file or open an existing Java file.
3. Make the changes to the contents of the Java file.
4. When you have finished working, click **File** → **Save** or **File** → **Save All** to save the file and retain all your changes.

Next:

You can now perform the following task:

- “Adding Java code dependencies”

Adding Java code dependencies

When you write your Java code for a JavaCompute node, you can include references to other Java projects and JAR files.

Before you start

To complete this task, you must have completed the following tasks:

- Add a “JavaCompute node” on page 542 to your message flow.
- “Creating Java code for a JavaCompute node” on page 286

The Java code in a JavaCompute node might contain references to other Java projects in your Eclipse workspace (internal dependencies), or to external JAR files, for example the JavaMail API (external dependencies). If other JAR files are referenced, you must add the files to the project class path.

1. Right-click the project folder of the project that you are working on and click **Properties**.
2. Click **Java Build Path** on the left pane.
3. Click the **Libraries** tab.
4. Perform one of the following steps:
 - To add an internal dependency, click **Add JARs...**, select the JAR file that you want to add then click **OK**.
 - To add an external dependency, click **Add External JARs...**, select the JAR file that you want to add, then click **Open**. Copy the file to *WorkPath*/shared-classes where *WorkPath* is the full path to the working directory of the broker. If you do not copy the external dependencies here `ClassNotFoundException` exceptions are generated at run time.

Tip:

The default value for *WorkPath* is one of the following values:

- For Windows systems, the default workpath is `c:\Documents and Settings\All Users\Application Data\IBM\MQSI`.
- For UNIX systems, the default workpath is `/var/mqsi`.
- For Linux systems, the default workpath is `/var/mqsi`.

You have now added a code dependency.

Deploying JavaCompute node code

The Message Brokers Toolkit handles the deploying of JavaCompute node code automatically. When you create a bar file and add the message flow, the Message Brokers Toolkit packages the compiled Java code and its dependencies into the bar file.

Writing Java

When you create a message flow, you include input nodes that receive the messages and, optionally, output nodes that send out new or updated messages. If the processing that must be performed on the message requires it, you can include other nodes after the input node that complete the actions that your applications need.

Some of the built-in nodes allow you to customize the processing that they provide. With a JavaCompute node you can provide Java code to control precisely the behavior of the node. This set of topics discusses how you can use Java to customize the JavaCompute node.

Using a JavaCompute node you can check and manipulate message content. You can:

- Read the contents of the input message
- Construct new output messages that are created from all, part, or none of the input message

Use the Debug perspective to debug a message flow that contains a JavaCompute node. When control passes to a JavaCompute node during debugging, the perspective opens the Java debugger, allowing you to step through the Java class code for the node.

This section provides more information about writing Java:

- “Manipulating message body data using a JavaCompute node”
- “Manipulating other parts of the message tree using a JavaCompute node” on page 297
- “Accessing broker properties from the JavaCompute node” on page 299
- “Accessing user-defined properties from a JavaCompute node” on page 300
- “Adding keywords to JAR files” on page 300
- “Interacting with databases using the JavaCompute node” on page 301
- “JavaCompute node Exception handling and the Failure terminal” on page 302
- “Logging errors with the JavaCompute node” on page 302
- Java user-defined node API

Manipulating message body data using a JavaCompute node

The message body is always the last child of root, and its parser name identifies it, for example XML or MRM.

The following topics describe how to refer to, modify, and create message body data. The information provided here is domain independent:

- “Accessing elements in a message tree from a JavaCompute node”
- “Transforming a message using a JavaCompute node” on page 291
- “Creating a simple filter using a JavaCompute node” on page 294
- “Propagating a message to the JavaCompute node Out and Alternate terminals” on page 295
- “Extracting information from a message using XPath 1.0 and a JavaCompute node” on page 295

Accessing elements in a message tree from a JavaCompute node:

When you want to access the contents of a message, for reading or writing, use the structure and arrangement of the elements in the tree that the parser creates from the input bit stream. Follow the relevant parent and child relationships from the top of the tree downwards, until you reach the required element.

The message tree is passed to a JavaCompute node as an argument of the evaluate method. The argument is a MbMessageAssembly object. MbMessageAssembly contains four message objects:

- Message
- Local Environment
- Global Environment

- Exception List

These objects are read-only, except for Global Environment. If you try to write to the read-only objects, a `MbReadOnlyException` is thrown.

This topic contains the following information about accessing elements in a message tree:

- “Traversing the element tree”
- “Accessing information about an element using a `JavaCompute` node” on page 291

Traversing the element tree:

This table shows the Java methods that can be used to access element trees, and the equivalent ESQL correlation name for each point in the tree.

Java accessor from <code>MbMessageAssembly</code>	ESQL correlation name
<code>getMessage().getRootElement()</code>	InputRoot
<code>getMessage().getRootElement().getLastChild()</code>	InputBody
<code>getLocalEnvironment().getRootElement()</code>	InputLocalEnvironment
<code>getGlobalEnvironment().getRootElement()</code>	Environment
<code>getExceptionList().getRootElement()</code>	InputExceptionList

The following methods can be used to traverse a message tree from an element of type `MbElement`:

getParent()

returns the parent of the current element

getPreviousSibling()

returns the previous sibling of the current element

getNextSibling()

returns the next sibling of the current element

getFirstChild()

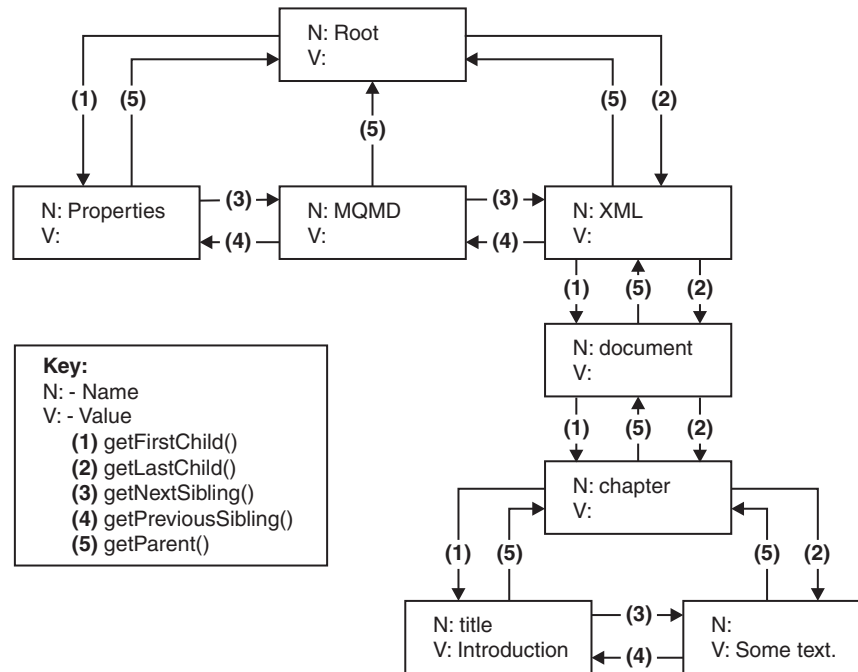
returns the first child of the current element

getLastChild()

returns the last child of the current element

The following example shows a simple XML message, and the logical tree that would be created from the message. The message has been sent using WebSphere MQ in this example. The logical tree diagram also shows the methods to call to navigate around the tree.

```
<document>
  <chapter title='Introduction'>
    Some text.
  </chapter>
</document>
```



The following Java code accesses the chapter element in the logical tree:

```

MbElement root = assembly.getMessage().getRootElement();
MbElement chapter = root.getLastChild().getFirstChild().getFirstChild();

```

Accessing information about an element using a JavaCompute node:

Use the following methods to return information about the referenced element. The Java user-defined node API provides further detail about these methods:

getName()

Returns the element name as a java.lang.String

getValue()

Returns the element value

getType()

Returns the generic type, which is one of the following types:

- NAME. An element of this type has a name, but no value.
- VALUE. An element of this type has a value, but no name.
- NAME/VALUE. An element of this type has both a value and a name.

getSpecificType()

Returns the parser-specific type of the element

getNamespace()

Returns the namespace URI of this element

Transforming a message using a JavaCompute node: These topics describe how to transform messages using a JavaCompute node:

- “Creating a new message using a JavaCompute node” on page 292
- “Copying a message using a JavaCompute node” on page 292
- “Setting, copying, and moving message elements using a JavaCompute node” on page 292

- “Creating new elements using a JavaCompute node” on page 293

Creating a new message using a JavaCompute node:

Many message transformation scenarios require a new outgoing message to be built. The *Create Message Class* template in the JavaCompute node wizard generates template code for this.

In the template code, the default constructor of MbMessage is called to create a blank message, as shown in the following Java code:

```
MbMessage outMessage = new MbMessage();
```

The headers can be copied from the incoming message using the supplied utility method, copyMessageHeaders(), as shown in this Java code:

```
copyMessageHeaders(inMessage, outMessage);
```

The new message body can now be created. First, add the top level parser element. For XML, this is:

```
MbElement outRoot = outMessage.getRootElement();
MbElement outBody = outRoot.createElementAsLastChild("XMLNSC");
```

The remainder of the message can then be built up using the createElement methods and the extended syntax of the broker XPath implementation.

Copying a message using a JavaCompute node:

The incoming message and message assembly are read-only. In order to modify a message, a copy of the incoming message must be made. The *Modifying Message Class* template in the JavaCompute node wizard generates this copy. The following copy constructors are called:

```
MbMessage outMessage = new MbMessage(inAssembly.getMessage());
MbMessageAssembly outAssembly = new MbMessageAssembly(inAssembly, outMessage);
```

The new outAssembly object is propagated to the next node.

Setting, copying, and moving message elements using a JavaCompute node:

This topic contains the following information about transforming a message:

- “Setting information about an element using a JavaCompute node”
- “Moving and copying elements using a JavaCompute node” on page 293

Setting information about an element using a JavaCompute node:

Use these methods to set information about the referenced element. The Java user-defined node API provides further detail about the methods:

setName()

Sets the name of the element

setValue()

Sets the value of the element

setSpecificType()

Sets the parser-specific type of the element

setNamespace()

Sets the namespace URI of the element

Moving and copying elements using a JavaCompute node:

You can use a JavaCompute node to copy or detach an element from a message tree using the following methods:

detach()

The element is detached from its parent and siblings, but any child elements are left attached

copy() A copy of the element and its attached children is created

There are four methods to attach an element or subtree that you have copied on to another tree:

addAsFirstChild(*element*)

Adds an unattached element as the first child of *element*

addAsLastChild(*element*)

Adds an unattached element as the last child of *element*

addBefore(*element*)

Adds an unattached element as the previous sibling of *element*

addAfter(*element*)

Adds an unattached element as the next sibling of *element*

Creating new elements using a JavaCompute node:

Use the following methods in a JavaCompute node to create new elements in a message tree:

- `createElementAsFirstChild()`
- `createElementAsLastChild()`
- `createElementBefore()`
- `createElementAfter()`

The method returns a reference to the newly-created element. Each method has three overloaded forms:

createElement...(int type)

Creates a blank element of the specified type. Valid generic types are:

- `MbElement.TYPE_NAME`. This type of element has only a name, for example an XML element.
- `MbElement.TYPE_VALUE`. This type of element has only a value, for example XML text that is not contained within an XML element.
- `MbElement.TYPE_NAME_VALUE`. This type of element has both a name and a value, for example an XML attribute.

Specific type values can also be assigned. The meaning of this type information is dependent on the parser. Element name and value information must be assigned using the `setName()` and `setValue()` methods.

createElement...(int type, String name, Object value)

Method for setting the name and value of the element at creation time.

createElement...(String parserName)

A special form of `createElement...()` that is only used to create top-level parser elements.

This example Java code adds a new chapter element to the XML example given in “Accessing elements in a message tree from a JavaCompute node” on page 289:

```

MbElement root = outMessage.getRootElement();
MbElement document = root.getLastChild().getFirstChild();
MbElement chapter2 = document.createElementAsLastChild(MbElement.TYPE_NAME,"Chapter",null);

// add title attribute
MbElement title2 = chapter2.createElementAsFirstChild(MbElement.TYPE_NAME_VALUE,
"title", "Message Flows");

```

This produces the following XML output:

```

<document>
  <chapter title="Introduction">
    Some text.
  </chapter>
  <chapter title="Message Flows"/>
</document>

```

Creating a simple filter using a JavaCompute node:

Before you start

To complete this task, you must have added a “JavaCompute node” on page 542 to your message flow.

The JavaCompute node has two output terminals, Out and Alternate. To use the JavaCompute node as a filter node, propagate a message to either the Out or Alternate terminal based on the message content. Use the JavaCompute node creation wizard to generate template code for a filter node:

Select the *Filtering Message Class* template in the JavaCompute node creation wizard to create a filter node.

The following template code is produced. It passes the input message to the Out terminal without doing any processing on the message.

```

public class jcn2 extends MbJavaComputeNode {

    public void evaluate(MbMessageAssembly assembly) throws MbException {
        MbOutputTerminal out = getOutputTerminal("out");
        MbOutputTerminal alt = getOutputTerminal("alternate");

        MbMessage message = assembly.getMessage();

        // -----
        // Add user code below

        // End of user code
        // -----

        // The following should only be changed
        // if not propagating message to the 'out' terminal

        out.propagate(assembly);
    }
}

```

The template produces a partial implementation of a method called evaluate(). The broker calls evaluate() once for each message that passes through the node. The parameter that is passed to evaluate() is the message assembly. The message assembly encapsulates the message that is passed on from the previous node in the message flow.

Add custom code to the template, and propagate messages to both the Out and Alternate terminals to create a message filter.

Propagating a message to the JavaCompute node Out and Alternate terminals:

The JavaCompute node has two output terminals, Out and Alternate. Therefore, you can use the node both as a filter node and as a message transformation node. After you have processed the message, propagate the message to an output terminal using a propagate() method. To propagate the message assembly to the Out terminal use the following method:

```
out.propagate(assembly);
```

To propagate the message assembly to the Alternate terminal, use the following method:

```
alt.propagate(assembly);
```

Extracting information from a message using XPath 1.0 and a JavaCompute node:

XPath is a query language designed for use with XML documents, but it can be applied to any tree structure for querying purposes. WebSphere Message Broker uses XPath to select elements from the logical message tree regardless of the format of the bit stream. For more information about XPath and the W3C definition of the XPath 1.0 standard, see XPath. The terminology used in this topic is based on the terminology used in the W3C definition of XPath 1.0.

This topic contains the following information:

- “Using the evaluateXPath method to extract message information”
- “XPath variable binding” on page 296
- “XPath namespace support” on page 296
- “Updating a message using XPath extensions” on page 297

Using the evaluateXPath method to extract message information

The evaluateXPath() method is included in the Java user-defined node API, and it supports XPath 1.0, with the following exceptions:

- Namespace axis and namespace node type. The namespace axis returns the actual XML namespace declaration nodes for a particular element. This allows you to manipulate XML prefix or URI declarations within an XPath expression. This axis returns an empty node set for bit streams that are not XML.
- If you use the id() function it throws an MbRecoverableException.

The evaluateXPath() method can be called on a MbMessage object (for absolute paths), or on a MbElement object (for relative paths). The XPath expression is passed to the method as a string parameter. A second form of this method is provided that takes an MbXPath object. This object encapsulates an XPath expression along with variable bindings and namespace mappings, if these are required.

The evaluateXPath() method returns an object of one of these four types, depending on the expression return type:

- java.lang.Boolean, representing the XPath Boolean type
- java.lang.Double, representing the XPath number type
- java.lang.String, representing the XPath string type

- `java.util.List`, representing the XPath node set. The List interface represents an ordered sequence of objects, in this case MbElements. It allows direct access to the elements, or the ability to get an Iterator or an MbElement array.

XPath variable binding

XPath 1.0 supports the ability to refer to variables inside an expression that have been assigned prior to evaluation. The MbXPath class has three methods for assigning and removing these variable bindings from user Java code. The value must be one of the four XPath 1.0 supported types:

- Boolean
- node set
- number
- string

XPath namespace support

For XML messages, namespaces are referred to using a mapping from an abbreviated namespace prefix to the full namespace URI as shown in the following XML example:

```
<ns1:aaa xmlns:ns1='http://mydomain.com/namespace1'
         xmlns:ns2='http://mydomain.com/namespace2'>
  <ns2:aaa>
    <ns1:bbb/>
  </ns2:aaa>
</ns1:aaa>
```

While the namespace prefix is convenient for representing the namespace, it is only meaningful within the document that defines that mapping. It is the namespace URI that defines the global meaning. Also, the concept of a namespace prefix is not meaningful for documents that are generated in a message flow because a namespace URI can be assigned to a syntax element without an xmlns mapping having been defined.

For this reason the XMLNS(C) and MRM parsers expose only the namespace URI to the broker and to user code (ESQL or plug-in code). ESQL allows the user to set up their own mappings in order to create abbreviations to these potentially long URIs. These mappings are not related in any way to the prefixes that are defined in the XML document (although they can be the same name).

The XPath processor allows you to map namespace abbreviations on to URIs that are expanded at evaluation time. The MbXPath class contains methods to assign and remove these namespace mappings. The XML example can be addressed using the following code:

```
MbMessage msg = assembly.getMessage();
List chapters= (List)msg.evaluateXPath("/document/chapter");
// this returns a list of all chapters in the document (length 1)

MbElement chapter = (MbElement)chapters.get(0); // the first one

// values can also be extracted directly using XPath
String title = (String)msg.evaluateXPath("string(/document/chapter/@title)");

String chapterText = (String)msg.evaluateXPath("string(/document/chapter/text())");
```


Updating a message using XPath extensions

The WebSphere Message Broker implementation of XPath provides the following extra functions for modifying the message tree:

set-local-name(*object*)

Sets the local part of the expanded name of the context node to the value specified in the argument. *object* can be any valid expression and is converted to a string as if a call to the string function were used.

set-namespace-uri(*object*)

Sets the namespace URI part of the expanded name of the context node to the value specified in the argument. *object* can be any valid expression and is converted to a string as if a call to the string function were used.

set-value(*object*)

This function sets the string value of the context node to the value specified in the argument. *object* can be any valid expression and is converted to a string as if a call to the string function were used.

To allow for syntax element trees to be built as well as modified, the following axis is available in addition to the 13 that are defined in the XPath 1.0 specification:

select-or-create::*name* or *?name*

?name is equivalent to `select-or-create::name`. If *name* is `@name`, an attribute is created or selected. This selects child nodes matching the specified `nametest` or creates new nodes according to the following rules:

- *?name* selects children called *name* if they exist. If there is not a child called *name*, *?name* creates it as the last child then selects it.
- *?\$name* creates *name* as the last child, then selects it.
- *?^name* creates *name* as the first child, then selects it.
- *?>name* creates *name* as the next sibling, then selects it.
- *?<name* creates *name* as the previous sibling, then selects it.

Manipulating other parts of the message tree using a JavaCompute node

The following topics describe how to access parts of the message tree other than the message body data. These parts of the logical tree are independent of the domain in which the message exists, and all these topics apply to messages in all supported domains, including the BLOB domain. You can access all parts of the message tree using a JavaCompute node, including the ExceptionList tree. Elements of the message tree can be accessed in the same way as the message body data, using a JavaCompute node.

- “Accessing headers using a JavaCompute node”
- “Updating the Local Environment with the JavaCompute node” on page 299
- “Updating the Global Environment with the JavaCompute node” on page 299

Accessing headers using a JavaCompute node:

If an input node receives an input message that includes message headers that the input node recognizes, the node invokes the correct parser for each header. Parsers are supplied for most WebSphere MQ headers. The topics listed below provide guidance for accessing the information in the MQMD and MQRFH2 headers that you can follow when accessing other headers that are present in your messages.

- “Copying message headers using a JavaCompute node” on page 298

- “Accessing the MQMD header using a JavaCompute node”
- “Accessing the MQRFH2 header using a JavaCompute node”

For further details of the contents of these and other WebSphere MQ headers for which WebSphere Message Broker provides a parser, see “Element definitions for message parsers” on page 735.

Copying message headers using a JavaCompute node: The *Modifying Message Class* template in the JavaCompute node wizard generates the following code to copy message headers using a JavaCompute node:

```
public void copyMessageHeaders(MbMessage inMessage, MbMessage outMessage) throws MbException
{
    MbElement outRoot = outMessage.getRootElement();
    MbElement header = inMessage.getRootElement().getFirstChild();

    while(header != null && header.getNextSibling() != null)
    {
        outRoot.addAsLastChild(header.copy());
        header = header.getNextSibling();
    }
}
```

Accessing the MQMD header using a JavaCompute node:

WebSphere MQ, WebSphere MQ Everyplace, and SCADA messages include an MQMD header. You can use a JavaCompute node to refer to the fields within the MQMD, and to update them.

The following Java code shows how to add an MQMD header to your message:

```
public void addMqmd(MbMessage msg) throws MbException
{
    MbElement root = msg.getRootElement();

    // create a top level 'parser' element with parser class name
    MbElement mqmd = root.createElementAsFirstChild("MQHMD");

    // specify next parser in chain
    mqmd.createElementAsFirstChild(MbElement.TYPE_NAME_VALUE,
        "Format",
        "XMLNS");
}
```

Accessing the MQRFH2 header using a JavaCompute node:

When you construct MQRFH2 headers in a JavaCompute node, there are two types of field:

- Fields in the MQRFH2 header structure (for example, Format and NameValueCCSID)
- Fields in the MQRFH2 NameValue buffer (for example mcd and psc)

The following code adds an MQRFH2 header to an outgoing message that is to be used to make a subscription request:

```
public void addRfh2(MbMessage msg) throws MbException
{
    MbElement root = msg.getRootElement();
    MbElement body = root.getLastChild();

    // insert new header before the message body
    MbElement rfh2 = body.createElementBefore("MQHRF2");
}
```

```

rfh2.createElementAsFirstChild(MbElement.TYPE_NAME_VALUE, "Version", new Integer(2));
rfh2.createElementAsFirstChild(MbElement.TYPE_NAME_VALUE, "Format", "MQSTR");
rfh2.createElementAsFirstChild(MbElement.TYPE_NAME_VALUE, "NameValueCCSID", new Integer(1208));

MbElement psc = rfh2.createElementAsFirstChild(MbElement.TYPE_NAME, "psc", null);
psc.createElementAsFirstChild(MbElement.TYPE_NAME, "Command", "RegSub");
psc.createElementAsFirstChild(MbElement.TYPE_NAME, "Topic", "department");
psc.createElementAsFirstChild(MbElement.TYPE_NAME, "QMgrName", "QM1");
psc.createElementAsFirstChild(MbElement.TYPE_NAME, "QName", "PUBOUT");
psc.createElementAsFirstChild(MbElement.TYPE_NAME, "RegOpt", "PersAsPub");
}

```

Updating the Local Environment with the JavaCompute node:

The Local Environment tree is part of the logical message tree in which you can store information while the message flow processes the message. The following information shows how to update the Local Environment:

1. Make a new copy of the local environment to update it. Use the full version of the copy constructor to create a new MbMessageAssembly object, as shown in the following example:

```

MbMessage env = assembly.getLocalEnvironment();
MbMessage newEnv = new MbMessage(env);

newEnv.getRootElement().createElementAsFirstChild(
    MbElement.TYPE_NAME_VALUE,
    "Status",
    "Success");

MbMessageAssembly outAssembly = new MbMessageAssembly(
    assembly,
    newEnv,
    assembly.getGlobalEnvironment(),
    assembly.getExceptionList(),
    assembly.getMessage());

getOutputTerminal("out").propagate(outAssembly);

```

2. Edit the copy to update the local environment.

Updating the Global Environment with the JavaCompute node:

The Global Environment tree is always created when the logical tree is created for an input message. However, the message flow neither populates it nor uses its contents. You can use this tree for your own purposes, for example to pass information from one node to another. You can use the whole tree as a scratchpad or working area.

The Global Environment can be altered across the message flow, therefore do not make a copy of it to alter. The following Java code shows how to alter the Global Environment:

```

MbMessage env = assembly.getGlobalEnvironment();
env.getRootElement().createElementAsFirstChild(MbElement.TYPE_NAME_VALUE, "Status", "Success");

getOutputTerminal("out").propagate(assembly);

```

Accessing broker properties from the JavaCompute node

For each broker, WebSphere Message Broker maintains a set of properties. You can access some of these properties from your Java programs. It can be useful, during the run time of your code, to have real-time access to details of a specific node, flow, or broker.

There are four categories of broker property:

- Those relating to a specific node
- Those relating to nodes in general
- Those relating to a message flow
- Those relating to the execution group

“Broker properties accessible from ESQL and Java” on page 983 includes a table that shows the groups of properties that are accessible from Java. The table also indicates if the properties are accessible from ESQL.

Broker properties:

- Are grouped by broker, execution group, flow, and node.
- Are case sensitive. Their names always start with an uppercase letter.
- Return NULL if they do not contain a value.

To access broker properties in a JavaCompute node, call methods on the following classes:

- MbBroker
- MbExecutionGroup
- MbMessageFlow
- MbNode

For example:

```
String brokerName = getBroker().getName();
```

Accessing user-defined properties from a JavaCompute node

To access broker user-defined properties from a JavaCompute node, use the `getUserDefinedAttribute(name)` method, where *name* is the name of the property that you are accessing. The type of the object that is returned depends on the type of the property that you are accessing. The object has one of a set of types:

- MbDate
- MbTime
- MbTimestamp
- Boolean
- byte[]
- String
- Integer 32-bit values
- Long 64-bit values
- Double
- BigDecimal
- BitSet

Adding keywords to JAR files

If a bar file contains JAR files, you can associate keywords with the JAR files.

1. Add a file called META-INF/keywords.txt to the root of the JAR file.
2. Add your keywords to the META-INF/keywords.txt file, because this file is parsed for keywords when it is deployed. Keywords have this format:

```
$MQSI keyword = value MQSI$
```

For example, a deployed bar file contains *compute.jar*, and *compute.jar* contains the file META-INF/keywords.txt with the following contents:

```
# META-INF/keywords.txt
$MQSI modified date = 3 Nov MQSI$
$MQSI author = john MQSI$
```

This content means that the keywords “modified date” and “author” are associated with the deployed file *compute.jar* in the Configuration Manager Proxy and in the Message Brokers Toolkit.

You have now added keywords to your JAR file.

Next:

When you have added keywords to your JAR file, you can see this information in the bar file editor.

Interacting with databases using the JavaCompute node

You can access databases from the JavaCompute node using only the following methods:

- MbSQLStatement
- Type 4 JDBC drivers

The broker resource manager does not coordinate database access when using type 4 JDBC drivers.

The MbSQLStatement class provides full transactional database access using ESQL. Create instances of this class using the createSQLStatement() method of MbNode, passing in the ODBC datasource, a broker ESQL statement, and optionally the transaction mode to the method.

- Calling select() on this object returns the results of the query.
- Calling execute() on this object executes a query where no results are returned, such as updating a table.

The following Java code shows how to access a database using MbSQLStatement:

```
MbMessage newMsg = new MbMessage(assembly.getMessage());
MbMessageAssembly newAssembly = new MbMessageAssembly(assembly, newMsg);

String table = "dbTable";

MbSQLStatement state = createSQLStatement( "dbName",
    "SET OutputRoot.XML.integer[] = PASSTHRU('SELECT * FROM " + table + "')");

state.setThrowExceptionOnDatabaseError(false);
state.setTreatWarningsAsErrors(true);
state.select( assembly, newAssembly );

int sqlCode = state.getSQLCode();
if(sqlCode != 0)
{
    // Do error handling here
}

getOutputTerminal("out").propagate(assembly);
```

JavaCompute node Exception handling and the Failure terminal

You do not need to catch exceptions that are thrown in a JavaCompute node. The broker handles exceptions automatically. If you catch an exception in your code, throw it again, allowing the broker to construct an exception list and propagate the message to the failure terminal, if one is connected. If you have not connected the failure terminal, the exception is thrown back to a Catch node or an input node.

Logging errors with the JavaCompute node

The MbService class contains a number of static methods for writing to the event log or syslog. Define message catalogs using Java resource bundles to store the message text. Three levels of severity are supported:

- information
- warning
- error

The JavaCompute Node sample samples demonstrate the use of resource bundles and logging.

Developing message mappings

Message mappings define the blueprint for creating a message. The topics in this section describe message mappings and explain how to develop them.

Concept topics:

- “Message mappings overview” on page 303
- “Message flows, ESQL, and mappings” on page 6

Task topics:

- “Creating message mappings” on page 306
- “Creating a message map file” on page 306
- “Configuring message mappings” on page 307
- “Adding a message to the source or target” on page 308
- “Adding a database to the source” on page 309
- “Mapping a target element from source message elements” on page 310
- “Mapping a target element from database tables” on page 311
- “Setting the value of a target element to a constant” on page 312
- “Setting the value of a target element using an expression or function” on page 313
- “Configuring conditional mappings” on page 314
- “Configuring mappings for repeating elements” on page 315
- “Configuring the LocalEnvironment” on page 316
- “Creating and calling submaps and subroutines” on page 316
- “Message mapping tips and restrictions” on page 324
- “Message mapping scenarios” on page 326

There is also a section of topics that contain reference information about message mapping:

- “Message mappings” on page 748
- “Message Mapping editor” on page 748

- “Mapping node” on page 756
 - “Migration of message mappings from Version 5.0” on page 759
- For a basic introduction to mapping, see the WebSphere Message Broker Basics IBM Redbook.

Message mappings overview

Message mappings define the blueprint for creating a message, where the created message is known as the target message. Messages can contain the following components:

- simple elements and attributes
- complex elements (structures)
- repeating simple or complex elements
- other (embedded) messages

Messages can contain protocol-specific headers, which might need to be manipulated by WebSphere Message Broker. Dynamic setting of a message destination (routing) within the WebSphere Message Broker might also be required. Values for target message elements can be derived from:

- input message elements (the input message is also known as the source message)
- database tables
- constant values
- functions supplied by the Mapping node
- user-defined functions

The logic to derive values can be simple or complex; conditional statements might be needed, as might loops, summations and other functions. All of the above mappings can be achieved using a Mapping node.

Find out more about message flows, ESQL, and mappings.

This section also contains information about “Advanced schema structures” on page 304.

Message flows, ESQL, and mappings

A message flow represents the set of actions performed on a message when it is received and processed by a broker. The content and behavior of a message flow is defined by a set of files that you create when you complete your definition and configuration of the message flow content and structure:

- The message flow definition file `<message_flow_name>.msgflow`. This is a required file and is created automatically for you. It contains details about the message flow characteristics and contents (for example, what nodes it includes, its promoted properties, and so on).
- The ESQL resources file `<message_flow_name>.esql`. This file is required only if your message flow includes one or more of the nodes that must be customized using ESQL modules. You can create this file yourself, or you can cause it to be created for you by requesting specific actions against a node.

You can customize the following built-in nodes by creating free-form ESQL statements that use the built-in ESQL statements and functions, and your own user-defined functions:

- Compute
- Database
- Filter

- The message mappings file <message_flow_name><_nodename>.msgmap. This file is required only if your message flow contains one or more of the nodes that must be customized using mappings. You can create this file yourself, or you can cause it to be created for you by requesting specific actions against a node. A different file is required for each node in the message flow that uses the Message Mapping editor.

You can customize the following built-in nodes by specifying how input values map to output values:

Node	Usage
“DataDelete node” on page 504	Use this node to delete one or more rows from a database table without creating an output message.
“DataInsert node” on page 507	Use this node to insert one or more rows in a database table without creating an output message.
“DataUpdate node” on page 510	Use this node to update one or more rows in a database table without creating an output message.
“Extract node” on page 513	Use this node to create a new output message that contains a subset of the contents of the input message. Use the Extract node only if no database is involved in the map.
“Mapping node” on page 567	Use this node to construct a new output message and populate it with information that is new, modified from the input message, or taken from a database. Use the Mapping node only if no database is involved in the map.
“Warehouse node” on page 661	Use this node to store all or part of a message in a database table without creating an output message.

You can use built-in ESQL functions and statements to define message mappings, and you can use your own ESQL functions.

Advanced schema structures

This section contains information about the following subjects:

- “Substitution groups”
- “Wildcards” on page 305
- “Derived types” on page 305
- “List types” on page 305
- “Union types” on page 305

Substitution groups: A substitution group is an XML Schema feature that provides a means of substituting one element for another in an XML message. The element that can be substituted is called the *head* element, and the substitution group is the list of elements that can be used in its place.

All possible substitutes of a head element are listed beneath the head element. You create mappings to or from members of substitution groups in the same way as you would map other elements.

An abstract head element of a substitution group is not displayed and when substitution is blocked, the substitution group folder is not displayed.

Wildcards: Any mapping that you perform to or from a wildcard results in a submap call. Specify the wildcard replacement when you choose the parameter of a submap call.

A wildcard element or attribute can be instantiated only with another element or attribute. The Message Mapping editor allows only a global element or attribute as a wildcard replacement.

Derived types: For an element of a given type, all known types that are based on it are shown in the Source and Target panes of the Message Mapping editor, and all attributes and elements of the base and derived types are listed under each type respectively.

You create mappings to or from a derived type and its contents in the same way that you would map any type or type content. When you map a derived type element, the Message Mapping editor generates ESQL code with the appropriate `xsi:type` attribute.

List types: A list type is a way of rendering a repeating simple value. The notation is more compact than the notation for a repeating element and provides a way to have multi-valued attributes.

You map list type attributes or elements in the same way that you would map any other simple type attribute or element. Mapping between two list type elements is the same as mapping between any two simple type elements.

To transform between a list type and a non-list type, such as a repeating element, write an ESQL function, then package the function as a map. The Message Mapping editor automatically selects this submap as the default transformation for the list type.

Union types: A union type is the same as a union of two or more other simple types and it allows a value to conform to any one of several different simple types.

Use the Message Mapping editor to create mappings to or from union type attributes or elements in the same way as you would map atomic simple type attributes or elements, as demonstrated in the following diagram:

```
<xsd:simpleType name="zipUnion">  
  <xsd:union memberTypes="USState listOfMyIntType"/>  
</xsd:simpleType>  
<xsd:element name=zip type=zipUnion/>
```

Related concepts

- Substitution groups in the message model
- Message model objects: wildcard attributes
- Message model objects: simple types
- “Message mappings overview” on page 303
- “Message flows, ESQL, and mappings” on page 6

Related tasks

- “Creating a new submap for a wildcard source” on page 317
- “Creating and calling submaps and subroutines” on page 316

Related reference

- Local element logical properties
- Wildcard element properties

Creating message mappings

The topics in this section describe how to create message mappings. Most actions can be achieved either by using the menu bar, or by right-clicking and choosing an action from a drop-down menu. For consistency, the following topics describe the menu bar method.

- “Creating a message map file”
- “Configuring message mappings” on page 307
- “Adding a message to the source or target” on page 308
- “Adding a database to the source” on page 309
- “Mapping from source: by selection or by name” on page 309
- “Mapping a target element from source message elements” on page 310
- “Mapping a target element from database tables” on page 311
- “Setting the value of a target element to a constant” on page 312
- “Setting the value of a target element using an expression or function” on page 313
- “Deleting a source or target element” on page 313
- “Configuring conditional mappings” on page 314
- “Configuring mappings for repeating elements” on page 315
- “Configuring the LocalEnvironment” on page 316
- “Creating and calling submaps and subroutines” on page 316
- “Editing a default-generated map manually” on page 322

There are also topics that discuss message mapping tips and restrictions, and message mapping scenarios.

Creating a message map file

Before you start:

1. Create a message flow project
2. Create a message flow
3. Define message flow content

To create a message map (.msgmap) file:

1. From the Broker Application Development perspective, click **File** → **New** → **Message Map**.

Alternatively, to create a message map from a message flow, open the message flow, right-click a node that supports mapping (such as a Mapping node or DataDelete node), and click **Open Map**.

The New Message Map wizard opens.

2. In the New Message Map wizard, specify the following criteria:
 - Whether you are going to map headers (after you have specified this, you cannot change it)
 - If you select the option **This map is called from a message flow node and maps properties and message body**, all input headers are copied to output automatically and you cannot specify the order of the headers.
 - If you select the option **This map is called from a message flow node and maps properties, headers, and message body**, you must map

explicitly the output headers that you want (see “Editing a default-generated map manually” on page 322 for more information).

- Whether the map is intended to be called from another map, which is known as a submap
- The source message or database
- The target message

After you have created a message map file, configure the message mappings.

Configuring message mappings

Use the Message Mapping editor to configure a message mapping. The editor provides the ability to set values for:

- the message destination
- message headers
- message content

See the “Mapping node” on page 567 topic for more information about how to set the properties of a Mapping node.

Wizards and dialog boxes are provided for tasks such as adding mappable elements and working with submaps. Mappings that are created with the Message Mapping editor are validated and compiled automatically, ready to be added to a broker archive (bar) file, and for subsequent deployment to WebSphere Message Broker.

Use the Message Mapping editor to perform the following tasks.

Common tasks:

- “Mapping a target element from source message elements” on page 310
- “Mapping a target element from database tables” on page 311
- “Setting the value of a target element to a constant” on page 312
- “Setting the value of a target element using an expression or function” on page 313
- “Configuring conditional mappings” on page 314
- “Configuring mappings for repeating elements” on page 315

Message destination tasks:

- “Configuring the LocalEnvironment” on page 316

Message content tasks:

- “Adding a message to the source or target” on page 308
- “Adding a database to the source” on page 309

Mapping from a message and database

Before you start:

1. Create a message flow project
2. Create a message flow
3. Define message flow content

The following instructions describe how to specify a message and a database as the data source.

1. Right-click a node that supports mapping, such as the Mapping node, and click **Open Map**.
2. Follow the on-screen instructions to complete the New Map wizard:
 - a. On the Select the map type pane, ensure that input message and database records are both selected.
 - b. On the Source and Target Mappables pane, select the message as the data source; do not select the database at this stage.
3. When the wizard completes, in the Spreadsheet pane, right-click \$target and click **Select Data Source**.
4. In the Select Database as Mapping Source wizard, select the database that you want to use and click **Finish**. A \$db:select entry appears in the Spreadsheet pane and the contents of the database table are displayed in the Source pane.
5. In order to map the Properties (messageSet, messageType, and messageFormat), expand the \$db:select and for entries in the Spreadsheet pane.
6. Right-click the \$target entry that appears below the for entry and click **Populate**. This populates the Spreadsheet pane with the top-level elements from the Target pane.
7. Right-click each of these elements and click **Populate**. Repeat this as many times as necessary for each level of element until all of the information from the Target pane appears in the Spreadsheet pane.
8. Perform mapping as usual.

Related tasks

“Creating a message map file” on page 306

“Configuring message mappings” on page 307

“Mapping a target element from source message elements” on page 310

“Mapping a target element from database tables” on page 311

“Scenario A: Mapping an airline message” on page 326

Adding a message to the source or target

Before you start:

Create a message map file.

To add a message to a source or target:

1. From the Message Mapping editor, click **Map** → **Add Sources and Targets**.
Alternatively, right-click in the Source, Target, or Spreadsheet pane and click **Add Sources and Targets**.
The Add Mappable wizard opens.
2. From the Add Mappable wizard, select messages from the message sets that are in your Message Brokers Toolkit workspace.
If one does not already exist, a project reference is created from your message flow project to the message set project that contains the selected messages.
A Mapping node can have only one source message, but can have several target messages. Therefore, you cannot add a source message if one already exists.

Adding a database to the source

To add a database table to your source mapping:

1. From the Message Mapping editor, click **Map** → **Add Sources and Targets**.

Alternatively, right-click in the Source, Target, or Spreadsheet pane and click **Add Sources and Targets**.

The Add Mappable wizard opens.

2. Select databases from the database definitions that are in your Message Brokers Toolkit workspace. When the wizard is complete:

- The Source pane contains a \$db:select entry.
- The Spreadsheet pane contains a \$db:select entry.

You cannot add a database table to the Target pane of a Mapping node or an Extract node.

Mapping from source: by selection or by name

There are two ways to map from source: by selection, or by name. The following steps describe how to map from source using the Map from Source wizard, or using the drag and drop method.

Using the Map from Source wizard

1. Select the source and target elements that you want to map by clicking them. (Ctrl+click to select multiple source or target elements.)

2. Click **Map** → **Map from Source**.

There are four possible scenarios: the first three scenarios represent mapping by selection, while the fourth scenario demonstrates mapping by name.

- If more than one mappable source element is selected, the selected sources are mapped to the selected target.
- If more than one mappable target element is selected, the selected source is mapped to the selected targets.
- If one mappable source and one mappable target are selected, and neither element has any children, the selected source is mapped to the selected target.
- If one mappable source and one mappable target are selected, and either element has children, the Map from Source wizard opens to allow you to perform mapping by name. Go to Step 3.

3. Choose the appropriate option from the Map from Source wizard:

- Deep copy entire complex element. This option copies the entire structure below the element. The option is available only when the selected source and target elements have the same type definition, or when the source type is derived from the target type.
- Map leaves. This option maps only the parts of the structure below the element that match each other.
- Map immediate children. This option maps only the immediate children of the source element to the immediate children of the target element that match each other. This option is available only when the selected source and target elements have immediate children that are mappable.

4. If you select the **Map leaves** or **Map immediate children** option, specify how names are matched.

Items of the same name are always considered to be a match and the **Map items of same names** check box is always selected by default. Two names are

considered to be the same if they contain the same alphanumeric characters in the same order. This comparison is not case sensitive, so FIRST_NAME and FirstName are considered to be a match.

If you select the **Map leaves** or **Map immediate children** option, the **Map items of similar names** check box is also available. Two names are considered to be similar if one name is a truncation of the other, such as first_name and name, or PART_NUMBER and partNum. Also, if one name is a contraction of another, such as November and nvmb, they are considered to be similar.

5. If you selected the **Map leaves** or **Map immediate children** option, verify that the mappings are relevant and edit them manually if necessary. You might need to delete unwanted mappings and add extra mappings.

Using the drag and drop method

Drag the appropriate source element or elements onto the target element or elements.

When you use the drag and drop method to map from source, mapping by selection is always performed. You can use the drag and drop method in the following scenarios:

- More than one mappable source element is selected. In this case, the selected sources are mapped to the selected target.
- More than one mappable target element is selected. In this case, the selected source is mapped to the selected targets.
- One mappable source and one mappable target are selected, and neither element has any children. In this case, the selected source is mapped to the selected target.
- The selected source can be deep copied. In this case, the entire structure below the element is copied. This option is available only when the selected source and target elements have the same type definition, or when the source type is derived from the target type.

Mapping a target element from source message elements

You can map:

- simple source elements to simple target elements
- source structures to target structures (where the source and target are of the same type)
- multiple simple source elements to a simple target element

The following sections describe how to perform mapping for these particular scenarios using the Message Mapping editor.

Mapping simple source elements to simple target elements

In the following example, the source element called Name does not contain the same children as the target element called Name:

Source	Target
Name Title First_name Middle_name Last_name	Name Title First_names Last_name

To map one of the child elements, drag the element from the Source pane onto the corresponding element in the Target pane; for example, drag the Last_name source element onto the Last_name target element.

The mapping is represented by a line between the source element and the target element and an entry for the mapping in Xpath format appears in the Spreadsheet pane. A triangular icon indicates which elements in the Source and Target panes have been mapped.

Mapping source structures to target structures

In the following example, the source element called Name has the same structure as the target element called Name:

Source	Target
Name Title First_name Middle_name Last_name	Name Title First_name Middle_name Last_name

To map the entire source structure to the target structure, drag the parent element (Name) from the Source pane onto the corresponding element (Name) in the Target pane. All the child elements are mapped.

Mapping multiple source elements to a simple target element

In the following example, you want to concatenate the First_name and Middle_name source elements to form a single target element called First_names:

Source	Target
Name Title First_name Middle_name Last_name	Name Title First_names Last_name

To map multiple source elements to a simple target element, Ctrl+click the appropriate source elements (First_name and Middle_name) and the target element (First_names), then click **Map** → **Map from Source**. A concatenate function appears in the Spreadsheet pane; you can edit this function to define how the concatenated target element looks, for example, by adding a white space between the two source elements.

To customize the target element (for example, to make the target value equal to the source value plus one), see “Setting the value of a target element using an expression or function” on page 313. You cannot map a simple element if one of its ancestors also has a mapping. For example, you cannot map Properties from source to target, then map Properties/MessageFormat.

Mapping a target element from database tables

To map a target element from a database table, set up the Mapping node to:

- retrieve the relevant rows from the database
- populate the message target elements with values from database

After you have added a database to the mapping, the Spreadsheet pane contains a \$db:select entry in the Map Script column. By default, its value is fn:true(), which means that all rows are retrieved from the database table. In database SQL, you

would restrict the number of rows by adding a WHERE clause to a database call; here is the equivalent method of restricting the number of rows in a Mapping node:

1. In the Spreadsheet pane, click the \$db:select row. This causes fn:true() to be put into the Edit pane.
2. Edit the expression in the Edit pane to specify the correct condition for the database call. To help you achieve this, you can:
 - a. Select any database columns that are relevant to the rows that are retrieved, and drag them from the Source pane to the Edit pane. These are the database column names that are used in the WHERE clause.
 - b. Select any source message elements with values that are relevant to the rows that are retrieved, and drag them from the Source pane into the Edit pane. These are values against which the selected database columns can be matched.
 - c. Open Content Assist by clicking **Edit** → **Content Assist**.
 - d. From Content Assist, select the functions to apply to message elements in the database call.

Here is an example of a \$db:select entry:

```
$db:select_1.BROKER50.JDOE.RESOLVEASSESSOR.ASSESSORTYPE = 'WBI' or $db:select_1.BROKER50.JDOE.RESOLVEASSESSOR.ASSESSORTYPE = $source/tns:msg_tagIA81CONF/AssessorType
```

A \$db:select entry retrieves all qualifying rows, so it is possible that more than one row is retrieved. By default, the selection is treated as repeating, which is indicated by the 'for' row immediately below \$db:select in the Spreadsheet pane. If you know that your database call will return only one row, you can delete this 'for' row.

After you have configured the \$db:select, populate the target message from the database by dragging the database column from the Source Pane to the message element in the Target pane. The mapping is indicated by a line between the database column in the Source pane and the element in the Target pane. An entry for this map in Xpath format also appears in the Spreadsheet pane. Triangular icons appear in the Source and Target panes next to objects that have been mapped.

Setting the value of a target element to a constant

To set the value of a target element to a constant:

1. In the Message Mapping editor Target pane, right-click the target element and click **Enter Expression**.
2. Enter the required constant in the Edit pane and click **Enter**. When entering the constant, observe the following criteria:
 - Enclose string element values in single quotes.
 - Enter numeric element values without quotes.

The Spreadsheet pane is updated with the value that you have defined.

You cannot set a value for a simple element if one of its ancestors also has a mapping. For example, you cannot map Properties from source to target, then set a value for Properties/MessageFormat.

Setting the value of a target element using an expression or function

There are two ways to set the value of a target element to an expression, depending on whether the target element has an entry in the Map Script column of the Message Mapping editor Spreadsheet pane:

- If the target element has an entry in the Map Script column:
 1. In the Spreadsheet pane, select the target element.
 2. Enter the required expression in the Edit pane.
 3. Press **Enter**.

The Spreadsheet pane is updated with the value or expression.

- If the target element does not have an entry in the Map Script column:
 1. In the Target pane, right-click the target element and click **Enter Expression**.
 2. Enter the required expression in the Edit pane.
 3. Press **Enter**.

The Spreadsheet pane is updated with the value or expression.

The following examples demonstrate techniques for entering mapping expressions in the Edit pane.

- If the target element is derived from a source element, drag the source element or elements onto the Edit pane; for example:
`$source/Properties/MessageSet`
- Use arithmetic expressions, such as:
`$source/Properties/Priority + 1`
- Use mapping, Xpath or ESQL function names. Content Assist (**Edit** → **Content Assist**) provides a list of available functions. For example:
`esql:upper($source/Properties/ReplyIdentifier)`
- You can perform casting in the Edit pane; for example:
`xs:string($source/Properties/CodedCharSetId)`

You cannot enter an expression for a simple element if one of its ancestors also has a mapping. For example, you cannot map Properties from source to target, then set a value of Properties/MessageFormat.

Deleting a source or target element

The following steps describe how to delete source and target elements using the "Message Mapping editor" on page 748:

- To delete a source path, modify the expression so that it no longer uses the source value to compute the target.
If this is the last use of the source path, the line linking the source and target is removed. If the expression no longer has any value, the target becomes unmapped.
- To delete a target from the Edit pane, click the target and click **Delete**.
The target structure is preserved if possible.
 - If you delete a "for" row, clicking **Delete** removes the single row.
 - If you delete a "condition" or "else" row, clicking **Delete**:
 - removes the entire block if there is at least one other "condition" or "else" row within the same "if" row

- removes the "if" row and the "condition" or "else" row, but preserves the content of the "condition" or "else" row when the selected "condition" or "else" row is the last one within the "if" row

Deleting the "if" row preserves the content of the last "condition" or "else" row within the "if" row and deletes everything else in the "if" row.

- To delete a database source, click the SELECT statement then remove all references to the source manually. Alternatively, delete the SELECT source in the Source pane then remove all references to the source manually.
- To delete a database target, delete the INSERT, UPDATE or DELETE statement. Alternatively, update or delete the statement in the Target pane.

Configuring conditional mappings

To set the value of a target element conditionally in a Mapping node:

1. In the Spreadsheet pane of the Message Mapping editor, select the target element and click **Map** → **If**.

Two rows are added to the Spreadsheet pane, above the target element:

- In the first row, Map Script is set to 'if'. You cannot enter anything in the Value column of this row.
- In the second row, Map Script is set to 'condition'. Its value is an expression that is evaluated to see whether it is true. If true, the target element is set to the value specified in its 'Value' column. Initially, its Value column is set to 'fn:true()', which means that the condition is always met, and the target element is always set to the Value column.

2. Change the expression in the condition row's Value column by selecting the cell, or the condition row, in the Spreadsheet pane and setting the value in the Edit pane.

Amend the expression in the Edit pane to specify the correct condition for the statement by performing the following steps:

- a. Select any database columns that are pertinent to the condition, and drag them from the Source pane into the Edit pane.
- b. Select any source message elements with values that are pertinent to the condition, and drag them from the Source pane into the Edit pane.
- c. Open Content Assist by clicking **Edit** → **Content Assist** and select the functions to be applied to the condition.

3. Add further conditions by selecting the condition row in the Spreadsheet pane, and clicking **Map** → **Condition**.

Two rows are added to Spreadsheet pane, below the target element:

- In the first row, Map Script is set to 'condition'. Process this as described in Step 2.
- In the second row, Map Script is set to the target element. Its Value cell is initially blank. Set this value as described in "Setting the value of a target element to a constant" on page 312, and "Setting the value of a target element using an expression or function" on page 313.

4. To set the value of a target element when the 'If' condition is not true, select the condition for the target element in the Spreadsheet pane, and click **Map** → **Else**.

Two rows are added to Spreadsheet pane, below the target element:

- In the first row, Map Script is set to 'else'. You cannot enter anything in the Value column of this row.

- In the second row, Map Script is set to the target element. Its value is initially blank. Set this value as described in “Setting the value of a target element to a constant” on page 312, and “Setting the value of a target element using an expression or function” on page 313.

Configuring mappings for repeating elements

To configure the Mapping node to process repeating elements, use the ‘For’ option in the Message Mapping editor Spreadsheet pane. The following combinations of repeating elements are possible:

- repeating source and non-repeating target
- non-repeating source and repeating target
- repeating source and repeating target

By default, if the source is a database, it is processed as a repeating source.

Configuring a repeating source and a non-repeating target:

To map a repeating source element to a non-repeating target element, drag elements between the Message Mapping editor Source and Target panes. The following items appear in the Spreadsheet pane:

- A ‘for’ row with Value set to the repeating source element.
- An ‘if’ row.
- A ‘condition’ row with Value set to `msgmap:occurrence($source/...) = 1`.
- A row with Map Script set to the target field and Value set to the source field.

The first occurrence of the source field is mapped to the target field. The ‘for’ row specifies that a loop is to be iterated for the specified repeating element. The ‘if’ and ‘condition’ rows restrict the logic to a single occurrence of the repeating element (see “Configuring conditional mappings” on page 314 for more information on conditional logic in a mapping node).

1. To map an occurrence other than the first, change the expression in the ‘condition’ row to `msgmap:occurrence($source/...) = n`, where *n* is the occurrence that you want to map.

If the repeating source field is within one or more repeating structures, a hierarchy of ‘for’, ‘if’, and ‘condition’ rows is placed in the Spreadsheet pane, one for each level of repetition.

2. If the source and target fields contain numeric data types, mapping all occurrences of a repeating source field to a non-repeating target results in the sum of all the source elements. Perform this mapping by selecting the source element and target element and clicking **Map** → **Accumulate**.

This action sets the following value in the Spreadsheet pane for the target element:

```
fn:sum($source/...)
```

You cannot map different occurrences of a repeating source element to different non-repeating target elements.

Configuring a non-repeating source and a repeating target:

To map a non-repeating source element to a repeating target element, drag elements between the Message Mapping editor Source and Target panes. The first occurrence of the target element is set to the value of the source element.

To map to an occurrence other than the first, complete the following steps:

1. If the target element is not shown in the Spreadsheet pane, right-click its lowest ancestor row, then click **Populate**. Repeat this action until the target element is shown.
2. Right-click the target element and click **Add Instance**. Repeat this action until you achieve the required number of occurrences. (Alternatively, add a number of instances by clicking **Add Group Instances**.)
3. Click the row of the occurrence that you want to set and set its value in the Edit pane. You can drag the source element from the Source pane to achieve this.

If you do not set a value for all occurrences, any occurrence for which a value is missing is not built into the target message.

Configuring a repeating source and a repeating target:

To map a repeating source element to a repeating target element drag elements between the Message Mapping editor Source and Target panes. The following items appear in the Spreadsheet pane:

- A 'for' row with Value set to the repeating source element.
- A row with Map Script set to the target field and Value set to the source field.

All occurrences of the source element are mapped to the respective occurrences of the target element. You can map repeating source structures to repeating target structures if the source and target are of the same complex type.

Configuring the LocalEnvironment

You can set values in the LocalEnvironment. If you set any values in the target LocalEnvironment, set the mapping mode property for the Mapping node to a value that contains LocalEnvironment. To do this, click **Properties** → **Basic** → **Mapping Mode**.

You cannot map Local Environment objects that are not listed.

Creating and calling submaps and subroutines

The following topics describe how to work with submaps and ESQL subroutines:

- "Creating a new submap"
- "Creating a new submap for a wildcard source" on page 317
- "Calling a submap" on page 318
- "Calling a submap from ESQL" on page 319
- "Calling an ESQL routine" on page 320
- "Creating and calling your own user-defined ESQL routine" on page 320

Creating a new submap:

This topic describes how to create a new submap. There are two ways to create a new submap:

- **Using File** → **New** → **Message Map**
 1. From the Broker Application Development perspective, click **File** → **New** → **Message Map**. The New Message Map wizard opens.
 2. Specify the project name and the name for the new submap.

3. Specify that the new map is a submap by selecting the option: **This map is called from another map and maps components within a message body.**
4. Specify the source and target elements.

The new submap opens in the Message Mapping editor.

- **Using Create new submap**

1. From the Broker Application Development perspective, open the message map for the required node.
2. In the Source pane, expand the tree and select the source.
3. In the Target pane, expand the tree and select the target.
4. Right-click either the source or target, then click **Create new submap.**

The new submap opens in the Message Mapping editor. If the original map file was called `simple_mapping.msgmap`, the new submap is called `simple_mapping_submap0.msgmap`.

Related concepts

“Message Mapping editor” on page 748

Related tasks

“Creating a message map file” on page 306

“Creating a new submap for a wildcard source”

“Calling a submap” on page 318

“Calling a submap from ESQL” on page 319

“Creating and calling submaps and subroutines” on page 316

Creating a new submap for a wildcard source:

This topic describes how to map a wildcard value in the source to a wildcard value in the target. You might expect a wild card in a Mapping node for example, when you are using a SOAP message (where the Body element contains a wildcard). This type of wildcard represents the payload of the message, where the payload is a message that is defined elsewhere in the message set. The submap can involve from 0 to n source wildcards and 0 or 1 target wildcards.

The “Message Mapping editor” on page 748 shows three kinds of wildcard, all of which allow you to create a submap:

Mapper construct	Message model construct	Choose concrete item for submap
Wildcard element	Wildcard element	Global element
Wildcard attribute	Wildcard attribute	Global attribute
Message with Wildcard Message child	Group with Composition of Message and Content Validation of Open or Open Defined	Message

1. Switch to the Broker Application Development perspective.
2. Open the message map for the required node.
3. In the Source pane, expand the tree and select the source wildcard.
4. In the Target pane, expand the tree and select the target wildcard.
5. Right-click either the source or the target wildcard, and click **Create new submap.** The Wildcard Specification wizard opens.

6. From the Wildcard Specification wizard, select the concrete item that will replace the source wildcard, according to the values shown in the table at the beginning of this topic.
7. Click **Next**.
8. From the Wildcard Specification wizard, select the concrete item that will replace the target wildcard, according to the values shown in the table at the beginning of this topic.
9. Click **Finish**.
10. Click **OK**. The submap opens in the Message Mapping editor.
11. From the submap, map the source message elements to the target message elements as required.
12. Click **OK**.

Calling a submap:

Use the Call Existing Submap wizard to call a submap. The submap must already be in the workspace.

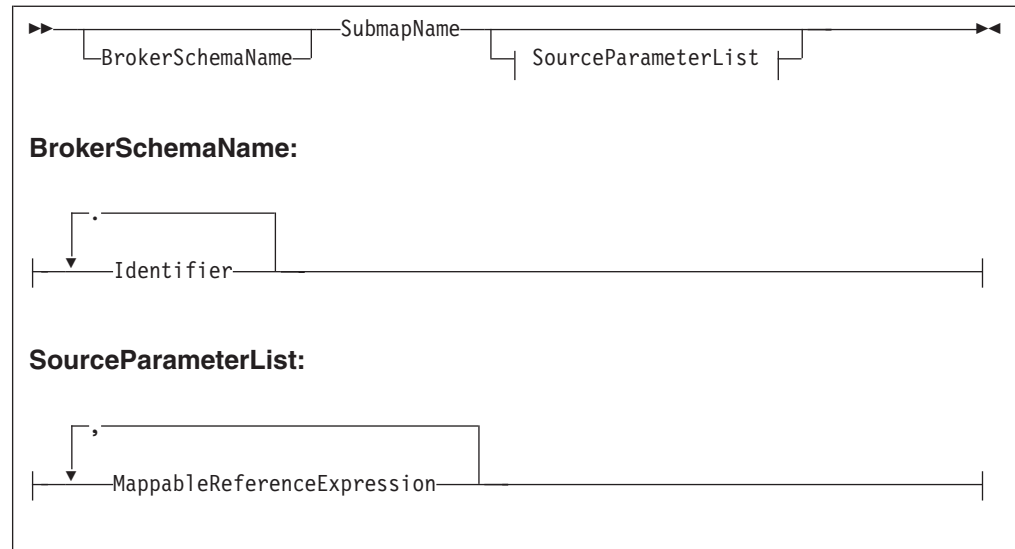
If a submap does not exist, use the **Create New Submap** menu option to create a submap that you can call. This action creates the new submap in the same folder as the calling map. It also allocates a default map operation name to the new submap. If the source or target in the calling map is a wildcard, a wizard allows you to choose a replacement element.

You can also map from a wildcard to a wildcard.

The following steps describe how to call a submap:

1. In the Broker Application Development perspective, open the calling map.
2. In the Source and Target panes, select one or more sources and one target. Any of the sources or the target can be a wildcard, an element, or an attribute.
3. Click **Map** → **Call Existing Submap**. The Call Existing Submap wizard opens.
4. Complete the wizard, following the on-screen instructions.

The call to the submap takes the following format:



Only source parameters appear in the call and only message parameters appear in the list.

Calling a submap from ESQL:

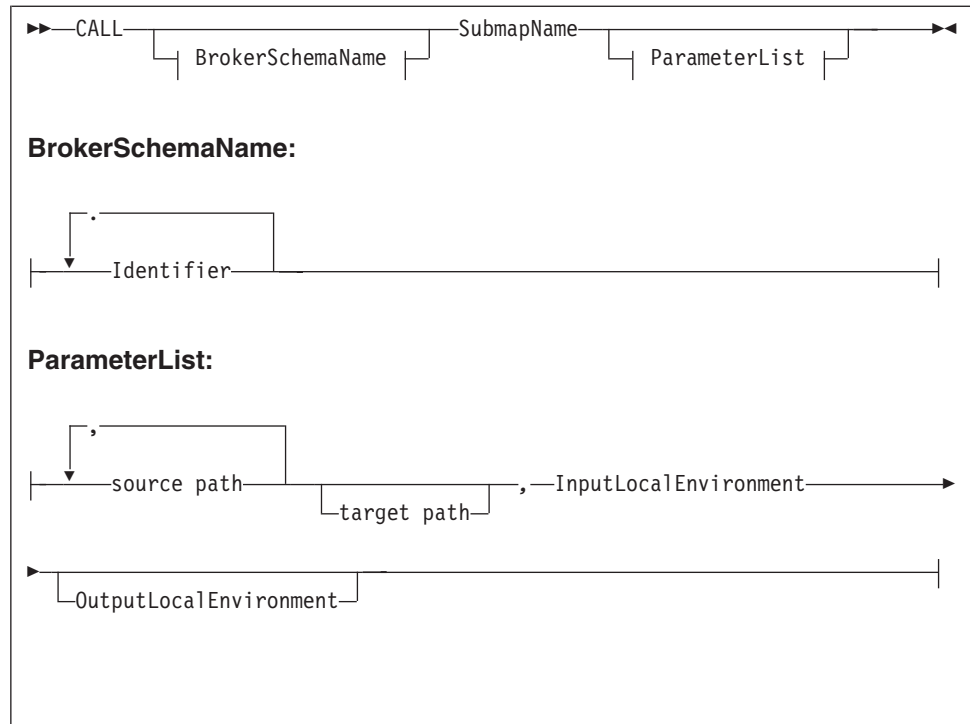
Before you start:

If a submap does not already exist, create one.

You can use the Message Mapping editor to perform mappings to a certain level of complexity. To create even more complex mappings, use ESQL. ESQL is particularly good for interacting with databases. The following steps describe how to call a submap from ESQL. Calling a submap from ESQL uses different parameters to when you call a submap from another map due to this extra level of complexity (when calling a submap from ESQL, the two local environment parameters are added at the end of the CALL statement).

1. Switch to the Broker Application Development perspective.
2. Right-click a node that supports ESQL and click **Open ESQL**. The ESQL file opens for that node.
3. Add a CALL statement to call a submap. Alternatively, press Ctrl+Space to access ESQL content assist, which provides a drop-down list that includes the submap name and expected parameters.

The following syntax diagram demonstrates the CALL statement:



Notes:

1. Only source parameters appear in the call and only message parameters appear in the list.
2. If the submap builds a message target, include the target path and `OutputLocalEnvironment` parameters. If the submap does not build a message target (for example, if the submap interacts with a database), these two parameters do not appear.

Related tasks

“Calling a submap” on page 318

“Creating a new submap” on page 316

“Creating a new submap for a wildcard source” on page 317

“Creating and calling submaps and subroutines” on page 316

Calling an ESQL routine:

To call an existing ESQL routine from a mapping, select the routine from the Call Existing ESQL Routine wizard. The ESQL routine must already exist in the workspace.

1. Switch to the Broker Application Development perspective.
2. Open the required mapping.
3. In the Source pane, select the required source.
4. In the Target pane, select the required target.
5. Right-click either the Source or Target pane and click **Call ESQL Routine**. The Call ESQL routine wizard opens.
6. Select the routine where the parameters and return types match the source and target selection.
7. Click **OK**.

Creating and calling your own user-defined ESQL routine:

For complex mappings, it is sometimes better to write an ESQL function that performs the work, then call the function from the Message Mapping editor. This topic describes a scenario where one of the output fields will be formed from a concatenation of input fields with additional text.

The message model used in this example is:

```
simple (message)
  int (xsd:int)
  str (xsd:str)
```

The ESQL function code used in this example is:

```
CREATE FUNCTION concatValues(IN val INTEGER, IN str CHAR) RETURNS CHAR
BEGIN
  return str || ' plus int val ' || CAST(val AS CHAR);
END;
```

1. Switch to the Broker Application Development perspective.
2. Right-click the Mapping node and click **Open Map**.
3. Accept the default Project and Name and click **Next**.
4. Accept the default usage and click **Next**.
5. Clear the **Based on records in a database** check box and click **Next**.
6. Select the source message `simple` and the target message `simple`, and click **Finish**.
7. In the Connection pane, open the source and target trees by clicking on the addition (+) symbols.
8. Open the `simple` trees on both sides in the same way.
9. Select `int` in the Source pane and drag it onto `int` in the Target pane. A line joins them to represent the mapping.
10. Right-click the message flow project that contains the message flow and the message map, and click **New** → **Message Flow ESQL File**.
11. Ensure that the name is the same as the name of the message flow, and click **Finish**.
12. Open the new ESQL file (for example, `flowname.esql`) and enter the example ESQL function code that is shown earlier in this topic.
13. Save the ESQL file.
14. In the Message Mapping editor Spreadsheet pane, select the Value column for the `str` item.
15. In the Edit pane, enter the function call. For example:
`esql:concatValues($source/simple/int, $source/simple/str)`
16. Save the mapping file by clicking **File** → **Save**.

Transforming a SOAP request message

SOAP is an XML based language defined by the W3C for sending data between applications. A SOAP message comprises an envelope containing:

- an optional header (containing one or more header blocks)
- a mandatory body.

For common envelope message formats, such as SOAP, where both the envelope and the messages that can appear within that envelope have to be modeled, use the Message Mapping editor to select from available messages at points in the model that are defined with **Composition="message"** and **Content validation="open"** or **"open defined"**.

Define the mappings by selecting from the allowed constituent messages. For example, in the case of SOAP, the outer level message is called Envelope and has one mandatory child element called Body, which is modeled with **Composition="message"**. If the permitted content of Body is modeled by separate messages Msg1 ... MsgN, define mappings for Envelope.Body.Msg1 and so on.

For complex type elements with type composition message, the Message Mapping editor follows these rules:

Content validation	Messages offered
Closed	Messages available in any message sets in the workspace
Open defined	Messages available in any message sets in the workspace
Open	The Message Mapping editor does not support open or open defined content when the type composition is NOT message

Mapping an embedded message

When you are working with type composition message, with content open or open-defined (and no children defined), map the embedded message using a submap:

1. In the main map, expand the levels (both source and target) of Envelope and Body until you find the wildcard message, and select this on both the source and target sides.
2. Right-click either the source or target and click **Create New Submap**.
3. From the dialog box, select a source (for example reqmess) and a target (for example rspmess).
4. With the submap open in the Message Mapping editor, make the appropriate mappings between the source (reqmess) and target (rspmess).

Editing a default-generated map manually

Sometimes, the map that is generated by the Message Mapping editor does not do everything that you want. If this is the case, there are a number of things that you can change manually. You can edit the structure directly by inserting, moving, copying, pasting, and deleting rows. The context menu provides a list of available editing actions with their keyboard equivalents. Here are some specific operations that you might want to perform:

- "Correcting out of order statements"
- "Creating message headers"
- "Creating conditional mappings" on page 323
- "Selecting the MQRFH2C compact parser" on page 324

Correcting out of order statements:

The Message Mapping editor does not validate the order of output elements against a schema definition so when it generates a map, the order of statements for elements in a sequence might be incorrect.

To fix this, move the statements so that they agree with the sequence defined in the schema.

Creating message headers:

When you create a map, if you select the option **This map is called from a message flow node and maps properties, headers and message body**, the map that is created allows additional elements, including MQ, HTTP, and JMS headers, to be mapped. When the mapping is created, the Message Mapping editor generates an "if" statement that contains a condition for each message header. You can edit this "if" statement to create the headers that you want.

- If you want one header, edit the condition of the header to be `fn:true()`.
Optional: You can also delete other unwanted "condition" blocks in the "if" statement.
- If you want more than one header, move the headers out of the "if" statement and delete the "if" statement.

If you use a Mapping node for a database to message mapping, and you select the option **This map is called from a message flow node and maps properties and message body**, the Message Mapping editor cannot generate an output MQMD header for the map file that is created. To ensure that an output MQMD header is created, perform one of the following steps:

- When you create the map file, select the option **This map is called from a message flow node and maps properties, headers and message body** and ensure that all mandatory fields in the MQMD header are set.
- If you want to use the **This map is called from a message flow node and maps properties and message body** option, add an additional source message to the map. The source message must be the same message as the intended target message. You do not need to create any mapping from the source message; the presence of the source message forces the Message Mapping editor to copy its MQMD header to the output message tree.

Creating conditional mappings:

When a mapping involves one of the following items:

- schema choice group
- derived type element
- substitution group member
- wildcard
- repeating element

the default mapping that is generated by the Message Mapping editor might be placed under a "condition" statement. If the condition is not what you had expected, edit the statements; here are the changes that you can make:

- Move statements in or out of a conditional block.
- Reorder conditions within an "if" statement.
- Create new conditions inside an "if" statement.
- Create new "if" statements.
- If you need only one of the elements in a choice group, set the condition expression for the element to `fn:true()` and delete the conditional blocks that contain the unwanted choice group members.
- When there are sibling or nested schema choices or sequence groups, and the default generated conditional statement does not reflect what you want, move the statements or re-create the "if" statements.

See the "Configuring conditional mappings" on page 314 topic for more information about conditional mappings.

Selecting the MQRFH2C compact parser:

If you are using a Compute node downstream from a Mapping node, and the Compute node produces an MQRFH2C field, select **Use MQRFH2C Compact Parser for MQRFH2 Domain** on the “MQInput node” on page 593. When you select this property, the MQRFH2C compact parser is used for MQRFH2 headers instead of the MQRFH2 parser.

Related concepts

“Message mapping tips and restrictions”

“Message flows, ESQL, and mappings” on page 6

Related tasks

“Creating a message map file” on page 306

“Configuring message mappings” on page 307

“Configuring conditional mappings” on page 314

“Accessing the MQRFH2 header” on page 190

Related reference

“Compute node” on page 492

“Mapping node” on page 567

Message mapping tips and restrictions

These tips assume that you have created a mapping node within the message flow, opened the Message Mapping editor, and selected both a source and a target message:

- “Source is a list and target is a list from source but with a new entry at the top of the list”
- “Change the target runtime parser”
- “Override the database schema name” on page 325
- “Map batch messages” on page 325
- “Mapping restrictions” on page 325

Source is a list and target is a list from source but with a new entry at the top of the list

1. Expand the target to display the element for which you want to create a new first instance. This might be a structure or a simple element.
2. Right-click the element and click **If**. A condition line appears immediately below.
3. Right-click the element and click **Copy**. Move to the condition line and click **Paste**. There are now two entries in the spreadsheet for your element.
4. Set the first of these entries to values of your choice. This is the first instance.
5. Right-click the second entry and click **For**. A for line appears in the spreadsheet.
6. Set the second entry to the value or values mapped from the source.
7. Set the for entry to the looping condition.
8. Click for, then drag the source field that represents the loop condition to the Expression editor.

Change the target runtime parser

When you first create a mapping, you nominate a message set for the target message. The parser that is associated with the output message is the runtime parser that is associated with the message set. For example, when a message set is

first created, the default runtime parser is MRM. This means that the Mapping node generates ESQL with the following format:

```
SET OutputRoot.MRM.Fielda...
```

If you change the runtime parser to XML or XMLNSC for example, the Mapping node generates ESQL with the following format:

```
SET OutputRoot.XMLNSC...
```

The source message's parser is determined by the MQRFH2 header or the input node. The Mapping node can handle all parsers on input. The Mapping node generates a target message with a parser that matches the runtime parser of the message set.

1. Open the message set file messageset.mset.
2. Change the runtime parser to a value of your choice, and save the message flow project or projects, referencing this message set for mapping purposes.
3. If you changed the parser to MRM, deploy the message set.
4. Deploy the message flow that contains the mappings and test your ESQL in a Compute node and other nodes to ensure that they still function as expected.

Override the database schema name

To change the database schema name that is generated in ESQL, use the **Override RDB schema** wizard in the **Specify Runtime Schema** dialog box. The default is the schema name of the database definitions that are imported into the Message Brokers Toolkit. Use this dialog box to change the value.

Map batch messages

You can configure a message mapping that sorts, orders and splits the components in a multipart message into a series of batch messages. These components can be messages or objects and they can have different formats. If this is the case, each component is converted and the message is reassembled before being forwarded.

1. Use a "RouteToLabel node" on page 630 in the message flow to receive multipart messages as input.

The RouteToLabel node is the next node in sequence after the "Mapping node" on page 567, and causes the flow to jump automatically to the specified label. You can specify a single RouteToLabel value in a splitting map, for all maps that output a message assembly. You can also use conditions to set the RouteToLabel value depending on the values in the source message.

2. Use the "Message Mapping editor" on page 748 to build maps that transform and propagate batch messages in a single node, without having to define an intermediate data structure.

Multipart messages can also contain repeating embedded messages, where each repeated instance of a message is propagated separately. Embedded messages must be from the same message set as the parent message.

Mapping restrictions

Unless stated explicitly, you can achieve required functionality by calling an ESQL function or procedure. Here are some restrictions:

- Mixed content fields cannot be mapped.
- Exceptions cannot be thrown directly in mapping nodes.

- Self-defined elements cannot be manipulated in mapping nodes (there is limited support for wildcards when the wildcards represent embedded messages).
- The Environment tree cannot be manipulated in the Mapping node.
- User variables cannot be defined and set.
- CASE expressions cannot be emulated; you must use if/else.
- Trees cannot be copied from input to output in order to modify elements within the copied tree. For example, the following ESQL cannot be modeled in a Mapping node:

```
SET OutputRoot.MQMD = InputRoot.MQMD; SET OutputRoot.MQMD.ReplyToQ = 'NEW.QUEUE';
```

You must set each field in the structure individually if you intend to modify one or more sibling fields.

Message mapping scenarios

This section contains some message mapping scenarios that demonstrate how to make the most of the message mapping functions:

- “Scenario A: Mapping an airline message”
- “Scenario B: Simple message enrichment” on page 334
- “Scenario C: Using a broker as auditor” on page 341
- “Scenario D: Complex message enrichment” on page 345
- “Scenario E: Resolving a choice with alternative message data” on page 363
- “Scenario F: Updating the value of a message element” on page 364

Scenario A: Mapping an airline message

This scenario demonstrates how to create, configure, and deploy a new message mapping. The message flow that is used in this example reads an XML input message (an airline message), then uses a Mapping node to achieve the following transformations:

- Convert the input message from XML to COBOL
- Modify an element in the input message from the results that are obtained by a database lookup
- Concatenate two elements in the input message to form a single element in the output message

Here are the steps that are involved in this scenario:

1. “Example names and values”
2. “Connect to the database and obtain the definition files” on page 328
3. “Create the message flow” on page 329
4. “Create the mapping file” on page 330
5. “Configure the mapping file” on page 330
6. “Deploy the mapping” on page 334

Now go to “Example names and values.”

Example names and values:

The table contains definitions of the example resources that are used in the mapping an airline message scenario:

Resource	Name	Definition
Airline state codes database table file	.tblxmi file	Contains the database table that holds the two-character airline state codes (for example, IL for Illinois)
Alias name	AIRLINEDBALIAS	Same as connection name and database name in this case
Broker archive (bar) file name	AIRLINE	Contains the message flow and message set projects, and the mapping file, and is deployed to the default execution group for the run time
COBOL copybook	AirlineRequest.cbl	Controls the structure of the COBOL output message
Connection name	AIRLINECONN	Same as alias name and connection name in this case
Database	AIRLINEDB	Contains the table XREF and is the same as the connection name and the alias name in this case
Database table (table tree)	XREF	Contains lookup information (in this case the two-code airline city code abbreviations STATE=Illinois, ABBREV=IL)
Default project	AIRLINE_MFP	The default message flow project. This is the project to which to copy the database definitions
Default queue manager	WBRK6_DEFAULT_QUEUE_MANAGER	The default queue manager that controls the message queue
ESQL select operation	\$db:select.AIRLINEDB.AIRLINE_SCHEMTREE.XREF.ABBERV	The ESQL select operation that performs a qualified database select operation
Input (XML) message	c:\airline\data\AirlineRequest.xml	The input message (in this case an XML message)
Input message source fields	FirstName,LastName	The source elements in the input message that are concatenated
Input queue name property	AIRLINE_Mapping_IN	The input queue
Mapping node rename	XML_TO_COBOL	the name of the node in the message flow that performs the mapping (the node was renamed from its default name)
Message mapping file name	AIRLINE.msgmap	The file that contains the mapping configuration used by the Mapping node
Message Set property	AIRLINE_MSP2	The message set project name
Message Type property	msg_AIRLINEREQUEST	The message type
Message Format	CWF1	The custom wire format (CWF) for COBOL output message
Message flow name	AIRLINE_Mapping	The name of the message flow
Message flow project	AIRLINE_MFP	The name of the message flow project
Message set projects	AIRLINE_MSP1,AIRLINE_MSP2	The names of the message set projects

Resource	Name	Definition
Msg Domain node property	MRM	The Message Repository Manager (MRM)
Msg Set Name node property	AIRLINE_MSP1	The message set name node property
Msg Type property	AirlineRequest	The message type property
Msg Format node property	XML1	The input message format
Output message target field	NAME	The result of the concatenation of FirstName and LastName in the input message. NAME is the element that is created in the output message.
Output queue name property	AIRLINE_Mapping_OUT	The output queue name
Resource folder	airline\resources	The folder where the mapping resources are stored
Schema tree	AIRLINE_SCHEMTREE	The name of the schema tree
Source	ABBREV	The source
Source tree	\$source/AirlineRequest	The source tree
Source message	AirlineRequest	The source message
Target	STATE	The target
Target message	AIRLINEREQUEST	The target message
Target tree	\$target/AIRLINEREQUEST	The target tree
XPath concatenation function	fn:concat(fn:concat(\$source/AirlineRequest/Purchase/Customer/FirstName, ' '), \$source/AirlineRequest/Purchase/Customer/LastName)	The XPath function that concatenates the two fields, FirstName and LastName

Now go to “Connect to the database and obtain the definition files.”

Connect to the database and obtain the definition files:

Before you start:

Create a message flow project.

This topic demonstrates how to define the database connection that allows the message flow to access the database table in order to look up the two-character airline state code for Illinois. The database is accessed from a Mapping node. The database must be defined to the Message Brokers Toolkit.

1. Switch to the Broker Application Development perspective.
2. Click **File** → **New** → **RDB Definition Files**. The Establish a connection to a database dialog box opens.
3. Set the connection name to the data source (for example AIRLINEDBALIAS) and click **Next**.
4. From the **Alias** drop-down menu, select the database alias name (for example AIRLINEDB) and accept the default user name and password.

5. Click **Test Connection**. A message in the **Test Connection** dialog box confirms that the connection was successful.
6. Click **Finish**.
7. From the Import RDB definition dialog box, click **Browse**.
8. Select the destination message flow project where the database definitions are to be copied (for example AIRLINE_MFP).
9. Click **OK**.
10. Ensure that the project field is set to the message flow project name (for example AIRLINE_MFP) and click **Finish**.
11. From the Resource Navigator, expand the message flow project (for example AIRLINE_MFP) to reveal the files that have been created. The database table that contains the set of two-character airline state codes is stored in a .tblxml file.

You have now defined the database to the mapping tools.

Now go to “Create the message flow.”

Create the message flow:

Before you start:

1. Create a message flow project.
2. “Connect to the database and obtain the definition files” on page 328.
3. Create a message flow by adding an MQInput node, and renaming the node (for example, to AIRLINE_Mapping_IN).
4. Set the queue name property (for example, to AIRLINE_Mapping_IN).
5. Add an MQOutput node to the message flow, and rename the node (for example, to AIRLINE_Mapping_OUT).

This topic demonstrates how to specify a message flow project, add a Mapping node, wire the nodes, and set the node properties.

1. Switch to the Broker Application Development perspective.
2. Open the message flow (for example, AIRLINE_Mapping) within the message flow project (for example, AIRLINE_MFP). This message flow forms the starting point for the mapping task.
3. Open the palette of nodes and add a Mapping node to the message flow. You might need to scroll down to find the Mapping node.
4. Rename the Mapping node (for example, to XML_TO_COBOL) by right-clicking the node and clicking **Rename**.
5. Wire the node terminals (for example, AIRLINE_Mapping_IN > XML_TO_COBOL > AIRLINE_Mapping_OUT).
6. Modify the properties of the MQInput node (for example, AIRLINE_Mapping_IN) by right-clicking the node and clicking **Properties**.
7. Click **OK**.
8. Modify the properties of the Mapping node (for example, XML_TO_COBOL).
9. Set the data source as the database name (for example, AIRLINEDBALIAS)
10. Click **OK**.

You have now created the required message flow, wired the nodes, and set the node properties.

Now go to “Create the mapping file.”

Create the mapping file:

Before you start:

Follow the instructions in these topics:

1. “Connect to the database and obtain the definition files” on page 328
2. “Create the message flow” on page 329

This topic demonstrates how to create a new mapping file, specify how it will be used, and specify the source and target mappable elements.

1. Switch to the Broker Application Development perspective.
2. From the message flow, right-click the node (for example XML_TO_COBOL) and click **Open Map**. The New Message Map wizard opens at the Create a New Map page.
3. Accept the default project (for example, AIRLINE_MFP) and name (for example, AIRLINE.msgmap), and click **Next**. The Select usage for the map page is displayed.
4. Accept the default usage (this map is called from a message flow node and maps properties) and click **Next**. The Select a map kind page is displayed.
5. Accept the default settings (the check boxes that are based on an input message and on records in a database are both selected) and click **Next**. The Source and target mappables page is displayed.
6. In the Message Mapping editor Source pane, select the source message (for example, AirlineRequest) from the first message set project (for example, AIRLINE_MSP1).
7. In the Target pane, select the target message (for example, AIRLINEREQUEST) from the second message set project (for example, AIRLINE_MSP2). Do not select the data source at this stage.
8. Click **Finish**.

You have now created the mapping file, defined its usage, and specified the source and target mappable elements.

Now go to “Configure the mapping file.”

Configure the mapping file:

Before you start:

Follow the instructions in these topics:

1. “Connect to the database and obtain the definition files” on page 328
2. “Create the message flow” on page 329
3. “Create the mapping file”

This set of topics demonstrates how to configure the mapping file by:

1. Specifying a data source
2. Mapping the message properties
3. Writing an XPath function that concatenates the elements in the input message
4. Specifying an ESQL select command

Now go to “Specify the data source.”

Specify the data source:

Before you start:

Follow the instructions in these topics:

1. “Connect to the database and obtain the definition files” on page 328
2. “Create the message flow” on page 329
3. “Create the mapping file” on page 330

This topic demonstrates how to specify the database to use as the source for the mapping.

1. From the Message Mapping editor Spreadsheet pane, select an item. The item that you select determines the scope of the \$db:select entry that is created by the action. For example, you can select \$target, an element, an attribute, a For condition, or another \$db:select entry. The Select database as mapping source dialog box opens.
2. Right-click and click **Select data source**.
3. From the Select Database as mapping source page, select a database (for example, AIRLINEDB) and click **Finish**. The Message Mapping editor adds the sources of the database table (for example, the XREF table) to the tree in the Message Mapping editor Source pane.

You have now added the data source to the Message Mapping editor Source pane.

Now go to “Map the message properties.”

Map the message properties:

Before you start:

Follow the instructions in these topics:

1. “Connect to the database and obtain the definition files” on page 328
2. “Create the message flow” on page 329
3. “Create the mapping file” on page 330
4. “Specify the data source”

This topic demonstrates how to map the message set, message type and message format properties.

1. From the Message Mapping editor Spreadsheet pane, expand the \$db:select entry and ‘for’ elements by clicking the addition (+) symbols to reveal the message properties.
2. Right-click \$target and click **Populate**.
3. Right-click Properties and click **Populate**.
4. From the Spreadsheet pane, click the value field of the message set, message type, and message format properties and modify each value. Use quotation marks because the values are string literals (without quotation marks, the values will be interpreted as XPath locations).
5. From the Target pane, open the \$target tree by clicking the addition (+) symbol, then open the properties within the \$target tree. In the Target pane, an orange

arrow is displayed next to the message set, message type and message format elements, indicating that their properties have been set.

6. From the Message Mapping editor Source pane, expand the properties for the \$source tree, and for each remaining property, map the source element to its corresponding target element by dragging from source to target. The blue arrows in the Message Mapping editor Source pane indicate that the sources elements have been mapped to targets elements.
7. Close the Properties tags in the source and target trees by clicking the subtraction (-) symbols, then expand the full trees of the source (for example, \$source/AirlineRequest) and the target (for example, \$target/AIRLINEREQUEST).
8. Save the map by clicking **File** → **Save**.

You have now mapped message set, message type and message format properties.

Now go to “Add the XPath concatenate function.”

Add the XPath concatenate function:

Before you start:

Follow the instructions in these topics:

1. “Connect to the database and obtain the definition files” on page 328
2. “Create the message flow” on page 329
3. “Create the mapping file” on page 330
4. “Specify the data source” on page 331
5. “Map the message properties” on page 331

This topic demonstrates how to write an XPath function that concatenates the FirstName and LastName from the input message, and adds a white space separator in the target NAME element. When you add the XPath expression and save the map, link lines are automatically generated between the source and target to indicate that these elements are mapped.

1. From the Message Mapping editor Source pane, select the first source to concatenate (for example, FirstName), Ctrl+click to select the second source to concatenate (for example, LastName), and drag both elements onto the target (for example, NAME) in the Target pane.
2. From the Message Mapping editor Spreadsheet pane, select the target (for example, NAME).
3. From the Edit pane, enter the XPath function (for example, `fn:concat($source/AirlineRequest/Purchase/Customer/FirstName, ' ', $source/AirlineRequest/Purchase/Customer/LastName)`
4. Save the map by clicking **File** → **Save**.

You have now added an XPath function that concatenates the two source elements in the input message into a single target element in the output message.

Now go to “Add the database Select operation.”

Add the database Select operation:

Before you start:

Follow the instructions in these topics:

1. "Connect to the database and obtain the definition files" on page 328
2. "Create the message flow" on page 329
3. "Create the mapping file" on page 330
4. "Specify the data source" on page 331
5. "Map the message properties" on page 331
6. "Add the XPath concatenate function" on page 332

This topic demonstrates how to add a database select operation that makes a qualified selection from the data source. In the spreadsheet value table, the \$db:select statement has the default value fn:true(), which returns all entries in the table. You must therefore replace this value with one that qualifies the selection, for example:

```
$db:select.LAB13STA.ARGOSTR.XREF.STATE=$source/AirlineRequest/Purchase/Customer/State
```

This example selects only the records from the database where the STATE column is the same as the State input field. The For entry is shown with the value \$db:select, which causes iteration over the values that are selected from the database. If you use an unqualified select, the following source to target mappings would exist:

Source	Target
Street	STREET
City	CITY
Zip	ZIP_CODE
FlightNumber	FLIGHT_NO
Date	TRAN_DATE
Price	COST
CreditCard	CC_NO
Status	STATUS1
Details	DETAILS

1. From the Message Mapping editor spreadsheet value table, replace the existing value fn:true() with the required value.
2. From the Message Mapping editor Source pane, open the \$db:select expression, then open the following trees:
 - a. the database tree (for example, AIRLINEDB)
 - b. the schema tree (for example, AIRLINE_SCHEMTREE)
 - c. the database table tree (for example XREF)
3. From the Message Mapping editor Source pane, select the source (for example, ABBREV), and drag it onto the target (for example, STATE) in the Target pane. Note that a connection line is not displayed initially.
4. From the Spreadsheet pane, you can see that the target field (for example, STATE) has been assigned a Select value (in this example, \$db:select.AIRLINEDB.AIRLINE_SCHEMTREE.XREF.ABBREV). The value identifies a specified column in the database table (in this example, the ABBREV column in the XREF table). Lines between the sources and targets now indicate that the mappings have occurred.
5. Save the mapping by clicking **File** → **Save**.

6. Save the message flow.
7. Check the Tasks pane to see if any errors have been generated.

You have now made a qualified selection from the database.

Now go to “Deploy the mapping.”

Deploy the mapping:

Before you start:

Follow the instructions in these topics:

1. “Connect to the database and obtain the definition files” on page 328
2. “Create the message flow” on page 329
3. “Create the mapping file” on page 330
4. “Configure the mapping file” on page 330

This topic demonstrates how to deploy the mapping to the run time by creating a broker archive (bar) file, which contains the message flow and message set projects (including the mapping file), then deploying the bar file to the default execution group.

1. From the Broker Administration perspective, right-click the project under the Broker Archives heading.
2. Click **New** → **Message Broker Archive**.
3. Name the bar file (for example, AIRLINE).
4. Click **Add to**. The Add to broker archive page is displayed.
5. Select the message flow project and message set projects that are used by this flow (for example, AIRLINE_MFP, AIRLINE_MSP1, AIRLINE_MSP2) and click **OK**. The projects are added to the bar file. A status indicator and message panel show when the process is complete.
6. Check to ensure that the required projects have been included in the bar file.
7. Save the bar file by clicking **File** → **Save**.
8. To deploy the bar file, right-click the bar file and click **Deploy File**. The Deploy a bar file page is displayed.
9. Select the default execution group, and click **OK**. A message in the Broker Administration message dialog box indicates successful deployment, and the deployed message flow project and message set projects appear in the Domains view. A message in the Event log also indicates successful deployment.

You have completed this scenario.

Scenario B: Simple message enrichment

This scenario demonstrates simple message enrichment and uses the Message Brokers Toolkit to create message flows and message sets, and to create and deploy broker archive (bar) files. The scenario also involves creating a Configuration Manager and a broker, and inputting instance messages that can contain MQRFH2 headers.

This scenario uses repeating instances and requires the following mapping functions:

- MRM in, MRM out (non-namespace)

- Map simple and Complex element source - target
- Map same element source - target
- Map different element source - target
- Map attribute source - target
- Map one sided element (edit mapping)
- Map one sided attribute (edit mapping)
- Perform arithmetic on numeric field mapping
- Map repeating simple element - single instance
- Map all instances of repeating simple element
- No MQRFH2 header

The names and values used for message flows, message sets, elements and attributes, and the expressions and code samples are for illustrative purposes only.

Here are the steps that are involved in this scenario:

1. “Develop a message flow and message model for simple and complex element mapping”
2. “Develop a message flow and message model for a target-only element” on page 337
3. “Develop a message flow and message model for dealing with repeating elements” on page 338
4. “Develop a message flow and message model for a simple message without an RFH2 header” on page 339

Develop a message flow and message model for simple and complex element mapping:

This is the first stage of the scenario to perform simple message enrichment. This topic demonstrates how to develop a message flow and message model for simple and complex element mapping, where there is the same source and target, a different source and target, or an attribute source and target. This task also involves changing field values and creating an instance document.

1. From the Broker Application Development perspective, create the following resources:
 - a. a message set project
 - b. a message set called MAPPING3_SIMPLE_messages. Ensure that the message set is namespace enabled with XML wire format.
 - c. a message definition file (no target namespace) called SIMPLE.
2. Create a message called addev1 that has the following structure:

```

addev1
  ssat      (xsd:string) local attribute
  ssel      (xsd:string) local element
  dsel1     (xsd:string) local element
  atel      local complex element
    latt    (xsd:string) attribute
  cell      local complex element
    intel   (xsd:int) local element
    strel   (xsd:string) local element
  dsel2     (xsd:string) global element
  cel2      (cel2ct) global complex type
    intel   (xsd:int) local element
    fltel   (xsd:float) local element

```

3. Create a message flow project called MAPPING3_SIMPLE_flows.

4. Create a message flow called addev1 that contains the following mapping: MQInput -> Mapping -> MQOutput.
5. Open the map in the Message Mapping editor and select message addev1 as both source and target
6. Expand all levels of both messages and wire the elements as shown:

```

ssat --- ssat
ssel --- ssel
dsel1 -- dsel2
latt ---- latt
cell --- cell
dsel2 -- dsel1
(cell2)
  intel ---- fltel
  fltel ---- intel

```

7. In the Spreadsheet pane, set the following expression:

```

dsel1 | esql:upper($source/addev1/dsel2)
@latt | esql:upper($source/addev1/atl/@latt)
(cell2)
  intel | $source/addev1/cel2/fltel + 10
  fltel | $source/addev1/cel2/intel div 10

```

8. Create an instance document with the appropriate RFH2 header and the following data:

```

<addev1 ssatt="hello">
<ssel>this</ssel>
<dsel1>first</dsel1>
<atel latt="attrib"/>
<cell>
<intel>2</intel>
<strel>1comp</strel>
</cell>
<dsel2>second</dsel2>
<cel2>
<intel>252</intel>
<fltel>3.89E+1</fltel>
</cel2>
</addev1>

```

You have created the following resources:

- message set MAPPING3_SIMPLE_messages, which you have populated with message addev1
- message flow addev1 in project MAPPING3_SIMPLE_flows, which contains the mapping addev1_Mapping.msgmap
- a file that contains an instance message

Now deploy the message set and message flow.

Deploy the message set and message flow:

This is the second stage of the scenario to perform simple message enrichment. This topic demonstrates how to deploy the message set and message flow and run the data through the broker.

1. Create a broker archive (bar) file called addev1.
2. Add the message set MAPPING3_SIMPLE_messages and the message flow addev1 to the bar file.
3. Deploy the bar file to the broker.
4. Put the instance document on the input queue.

The output message looks like this:


```

<addev1 ssat="hello">
<ssel>this</ssel>
<dse1>SECOND</dse1>
<atel latt="ATTRIB"/>
<cel1>
<intel>2</intel>
<strel>lcomp</strel>
</cel1>
<dse2>first</dse2>
<cel2>
<intel>48</intel>
<fltel>2.5E+1</fltel>
</cel2>
</addev1>

```

Now go to “Develop a message flow and message model for a target-only element.”

Develop a message flow and message model for a target-only element:

Before you start

Perform the steps in the following topic:

1. “Develop a message flow and message model for simple and complex element mapping” on page 335

This is the third stage of the scenario to perform simple message enrichment. This topic demonstrates how to develop a message flow and message model for a target-only element. It also involves attributing a mapping and creating an instance document.

1. Create a message called addev2, which has the following structure:

```

addev2
  matt          (xsd:string) local attribute
  ssel          (xsd:string) local element
  csel          local complex element
  elatt        (xsd:string) local attribute

```

2. Create a second message called trigger, which has the following structure:

```

trigger
  start          (xsd:string) local element

```

3. Create a message flow called addev2, which contains the following mapping: MQInput -> Mapping -> MQOutput.
4. Open the map and select trigger as the source and addev2 as the target.
5. In the Spreadsheet pane, expand the target message fully and set the target fields as shown:

```

matt   | 'first attribute'
ssel   | 'string element'
elatt  | 'second attribute'

```

6. Expand the Properties folder in the Spreadsheet pane and set the following value:

```

MessageType | 'addev2'

```

7. Create an instance document with the appropriate RFH2 header and the following data:

```

<trigger>
<start>yes</start>
</trigger>

```

You have created the following resources:

- two messages called addev2 and trigger
- a message flow called addev2, which contains the mapping addev2_Mapping.msgmap
- a file that contains an instance message

Now deploy the message set and message flow.

Deploy the message set and message flow:

This is the fourth stage of the scenario to perform simple message enrichment. This topic demonstrates how to deploy the message set and message flow and run the data through the broker.

1. Create a broker archive (bar) file called addev2.
2. Add the message set MAPPING3_SIMPLE_messages and the message flow addev2 to the bar file.
3. Deploy the bar file to the broker.
4. Put the instance document on the input queue.

The output message looks like this:

```
<addev2 matt="first attribute">
<ssel>string element</ssel>
<csel elatt="second attribute"></csel>
</addev2>
```

Now go to “Develop a message flow and message model for dealing with repeating elements.”

Develop a message flow and message model for dealing with repeating elements:

Before you start

Perform the steps in the following topics:

1. “Develop a message flow and message model for simple and complex element mapping” on page 335
2. “Develop a message flow and message model for a target-only element” on page 337

This is the fifth stage of the scenario to perform simple message enrichment. This topic demonstrates how to develop a message flow and message model for dealing with repeating elements, a single instance and all instances.

1. Create a message called addev3, which has the following structure:

```
addev3
  frepstr      (xsd:string) local element, minOccurs=3, maxOccurs=3
  vrepstr      (xsd:string) local element, minOccurs=1, maxOccurs=4
  urepstr      (xsd:string) local element, minOccurs=1, maxOccurs=-1
```

2. Create a message flow called addev3, which contains the following mapping: MQInput -> Mapping -> MQOutput.
3. Open the map and select addev3 as both source and target
4. In the upper pane, map each source to the corresponding target, as illustrated in this example:

```
frepstr --- frepstr
vrepstr --- vrepstr
urepstr --- urepstr
```

5. In the Spreadsheet pane, expand fully the target addev3.

6. Highlight and delete the For item above the vrepstr entry.
7. Create an instance message with the appropriate RFH2 header and the following data:

```
<addev3>
<frepstr>this</frepstr>
<frepstr>that</frepstr>
<frepstr>other</frepstr>
<vrepstr>only one</vrepstr>
<vrepstr>extra</vrepstr>
<urepstr>first</urepstr>
<urepstr>second</urepstr>
<urepstr>third</urepstr>
<urepstr>fourth</urepstr>
<urepstr>fifth</urepstr>
</addev3>
```

You have created the following resources:

- a message called addev3
- a message flow called addev3, which contains the mapping addev3_Mapping.msgmap
- a file that contains an instance message

Now deploy the message set and message flow.

Deploy the message set and message flow:

This is the sixth stage of the scenario to perform simple message enrichment. This topic demonstrates how to deploy the message set and message flow and run the data through the broker.

1. Create a broker archive (bar) file called addev3.
2. Add the message set MAPPING3_SIMPLE_messages and the message flow addev3 to the bar file.
3. Deploy the bar file to the broker.
4. Put the instance document on the input queue.

The output message looks like this:

```
<addev3>
<frepstr>this</frepstr>
<frepstr>that</frepstr>
<frepstr>other</frepstr>
<vrepstr>only one</vrepstr>
<urepstr>first</urepstr>
<urepstr>second</urepstr>
<urepstr>third</urepstr>
<urepstr>fourth</urepstr>
<urepstr>fifth</urepstr>
</addev3>
```

Now go to “Develop a message flow and message model for a simple message without an RFH2 header.”

Develop a message flow and message model for a simple message without an RFH2 header:

Before you start

Perform the steps in the following topics:

1. "Develop a message flow and message model for simple and complex element mapping" on page 335
2. "Develop a message flow and message model for a target-only element" on page 337
3. "Develop a message flow and message model for dealing with repeating elements" on page 338

This is the seventh stage of the scenario to perform simple message enrichment. This topic demonstrates how to develop a message flow and message model for a simple message without an RFH2 header.

1. Create a message set project and a message set called MAPPING3_SIMPLE_xml.
2. On the message set parameters page, set the Runtime Parser to XML.
3. Create a message definition file called SIMPLE.
4. Create a message called addev4, which has the following structure:

```
addev4
  str1          (xsd:string) local element
  cel           local complex element
    int1        (xsd:int) local element
    bool1       (xsd:boolean) local element
```

5. Create a message flow called addev4, which contains the following mapping: MQInput -> Mapping -> MQOutput.
6. On the MQInput node Default properties page, set the Message Domain to XML.
7. Open the map and select addev4 as both source and target.
8. Map the inputs to the corresponding outputs, as shown in this example:

```
str1 --- str1
int1 --- int1
bool1 --- bool1
```

9. Create an instance message with no RFH2 header and the following data:

```
<addev4>
<str1>this</str1>
<cel>
<int1>452</int1>
<bool1>0</bool1>
</cel>
</addev4>
```

You have created the following resources:

- a message set called MAPPING3_SIMPLE_xml, which contains the message addev4
- a message flow called addev4, which contains the mapping addev4_Mapping.msgmap
- a file that contains an instance message

Now deploy the message set and message flow.

Deploy the message set and message flow:

This is the final stage of the scenario to perform simple message enrichment. This topic demonstrates how to deploy the message set and message flow and run the data through the broker.

1. Create a broker archive (bar) file called addev4.
2. Add the message flow called addev4 to the bar file.

3. Deploy the bar file to the broker.
4. Put the instance document on the input queue.

The output message looks like this:

```
<addev4>
<str1>this</str1>
<cel>
<int1>452</int1>
<bool1>0</bool1>
</cel>
</addev4>
```

You have completed the scenario.

Scenario C: Using a broker as auditor

This scenario demonstrates how to use a broker as an auditor and uses the Message Brokers Toolkit to create message flows and message sets, and to create and deploy broker archive (bar) files. It also involves creating a Configuration Manager and a broker, and inputting instance messages that can contain MQRFH2 headers.

The scenario uses database updates that have been defined using mappings. The broker receives a confirmation for a provisional booking, the message flow inserts a row into a database table representing the confirmation, updates a counter in another table representing the key of the confirmation, and deletes the provisional booking from a third table.

This scenario uses the DataDelete, DataInsert and DataUpdate nodes in the message flow, and requires the following mapping functions:

- Mapping in DataInsert node
- Combine input data into single insert
- Mapping in DataUpdate node
- Mapping in DataDelete node
- Bar file to override datasource

The names and values used for message flows, message sets, elements and attributes, and the expressions and code samples are for illustrative purposes only.

Here are the steps that are involved in this scenario:

1. "Develop a message flow"
2. "Deploy the message set and message flow" on page 343
3. "Override the data source of one of the nodes" on page 344
4. "Create a bar file, edit the configuration and deploy" on page 344

Develop a message flow:

This is the first stage of the scenario to use a broker as an auditor. This topic demonstrates how to develop a message flow to map several fields of input data into a single insert record for a database. It also involves updating another table and deleting a third table, as well as developing corresponding message models and instance messages.

1. Create a database called MAPDB and a table called CONFIRMATION, which contains the following columns:

RESID	INTEGER
-------	---------

2. Populate the CONFIRMATION table with the value shown:
9052
3. Create another table called RESERVATION, which contains the following columns:

RESID	INTEGER
NAME	VARCHAR(20)
PARTY	INTEGER
PAYMENT	DECIMAL(8,2)
4. Populate the RESERVATION table with the values shown:
8214, 'ARCHIBALD', 2, 0.0
2618, 'HENRY', 4, 120.0
9052, 'THAW', 3, 85.0
5. Create another table called PROVISIONAL, which contains the following columns:

RESID	INTEGER
-------	---------
6. Populate the PROVISIONAL table with the values shown:
8214 2618
7. Create a Windows ODBC Data Source Name for the database, and register the database with the Configuration Manager by clicking **File** → **New** → **RDB Definitions File**.
8. Create a message set project and a message set called MAPPING3_AUDIT_messages (ensuring that the message set is namespace enabled, with XML wire format) and create a message definition file called AUDIT.
9. Create a message called addev1, which has the structure:


```

addev1
  id          (xsd:int) local element
  status      (xsd:string) local element
  name        (xsd:string) local element
  size        (xsd:int) local element
  payment     (xsd:decimal) local element
      
```
10. Create a message flow project called MAPPING3_AUDIT_flows.
11. Create a message flow called addev1, which contains the following mapping: MQInput -> DataInsert -> DataUpdate -> DataDelete -> MQOutput.
12. For the DataInsert node, set the Data Source property to MAPDB.
13. Open the mapping for the DataInsert node and select MAPPING3_AUDIT_messages addev1 as the source, and MAPDB.SCHEMA.CONFIRMATION as the target.
14. Wire the source to the target as shown:


```

addev1          MAPDB
  id ----- RESID
      
```
15. For the DataUpdate node, set the Data Source property to MAPDB.
16. Open the mapping for the DataUpdate node and select MAPPING3_AUDIT_messages addev1 as the source, and MAPDB.SCHEMA.RESERVATION as the target.
17. Wire the source to the target as shown:


```

addev1          MAPDB
  id ----- RESID
  name ----- NAME
  size ----- PARTY
  payment ----- PAYMENT
      
```

18. In the Spreadsheet pane, select \$db:update and change fn:true() to \$db:update.MAPDB.MQSI.RESERVATION.RESID = \$source/addev1/id and \$source/addev1/status = 'CONFIRM'.
19. For the DataDelete node, set the Data Source property to MAPDB.
20. Open the mapping for the DataDelete node and select MAPPING3_AUDIT_messages addev1 as the source, and MAPDB.SCHEMA.PROVISIONAL as the target.
21. In the Spreadsheet pane, select \$db:delete and change fn:false() to \$db:delete.MAPDB.MQSI.PROVISIONAL.RESID = \$source/addev1/id.
22. Create the following instance message with appropriate RFH2 headers:


```
<addev1>
<id>8214</id>
<status>CONFIRM</status>
<name>ARCHIBALD</name>
<size>2</size>
<payment>1038.0</payment>
</addev1>
```

You have created the following resources:

- a message set called MAPPING3_AUDIT_messages, which is populated with the message addev1
- a message flow called addev1 in project MAPPING3_AUDIT_flows, which contains the mapping files addev1_DataInsert.msgmap, addev1_DataUpdate.msgmap, and addev1_DataDelete.msgmap
- the database MAPDB with populated tables CONFIRMATION, RESERVATION, and PROVISIONAL
- a file that contains an instance message for test.

Now go to “Deploy the message set and message flow.”

Deploy the message set and message flow:

Before you start

Perform the steps in the topic “Develop a message flow” on page 341.

This is the second stage of the scenario to use a broker as an auditor. This topic demonstrates how to deploy the message set and message flow and run the instance messages through the broker.

1. Create a bar file called addev1.
2. Add the message set MAPPING3_AUDIT_messages and the message flow addev1 to the bar file.
3. Deploy the bar file to the broker.
4. Put the instance document on the input queue.

The output messages are the same as the input. Database table contents look like this:

```
CONFIRMATION
RESID
```

```
-----
      9052
      8214
```

```
RESERVATION
RESID      NAME                PARTY      PAYMENT
```

```

-----
      8214 ARCHIBALD                2    1038.00
      2618 HENRY                    4     120.00
      9052 THAW                      3      85.00

PROVISIONAL
RESID
-----
      2618

```

Now go to “Override the data source of one of the nodes.”

Override the data source of one of the nodes:

Before you start

Perform the steps in the following topics:

1. “Develop a message flow” on page 341
2. “Deploy the message set and message flow” on page 343

This is the third stage of the scenario to use a broker as an auditor. This topic demonstrates how to override the data source of one of the nodes by changing the configuration of its broker archive (bar) file.

1. Create a database called ALTDB, and a table called CONFIRMATION, which contains the following columns:

RESID	INTEGER
-------	---------
2. Create a Windows ODBC Data Source Name for the database, then register the database with the Configuration Manager by clicking **File** → **New** → **RDB Definitions File**.

You have created a database called ALTDB with a table called CONFIRMATION.

Now go to “Create a bar file, edit the configuration and deploy.”

Create a bar file, edit the configuration and deploy:

Before you start

Perform the steps in the following topics:

1. “Develop a message flow” on page 341
2. “Deploy the message set and message flow” on page 343
3. “Override the data source of one of the nodes”

This is the final stage of the scenario to use a broker as an auditor. This topic demonstrates how to create a broker archive (bar) file, edit the configuration and deploy.

1. Add the message flow addev1 to the bar file again.
2. Select the Configure tab of the bar file editor and click the DataInsert icon.
3. Change the Data Source field from MAPDB to ALTDB, and save the bar file.
4. Deploy the bar file to the broker.
5. Put the instance document on the input queue.

The output message is the same as the input. In the ALTDB database the table contents look like this:

You have completed this scenario.

Scenario D: Complex message enrichment

This scenario demonstrates complex message enrichment and uses complex message manipulation. Use the Message Brokers Toolkit to create message flows and message sets, and to create and deploy broker archive (bar) files. The scenario also involves creating a Configuration Manager and a broker, and inputting instance messages that can contain MQRFH2 headers.

This scenario requires the following mapping functions:

- MRM in, MRM out (namespace)
- Other nodes required to complete message
- Conditional mapping
- CASE mapping (both syntax formats)
- If/condition
- Combining multiple source fields into a single target field (inter namespace)
- Nested repeating complex and simple elements
- Target data derived from database
- String, numeric, datetime functions
- User-defined ESQL procedures and functions
- User-defined Java routines

The names and values used for message flows, message sets, elements and attributes, and the expressions and code samples are for illustrative purposes only.

Here are the steps that are involved in this scenario:

1. "Develop a message flow that contains other nodes"
2. "Develop a message flow to map target fields from multiple other fields" on page 348
3. "Develop a message flow and message model for mapping a complex nested, repeating message" on page 350
4. "Develop a message flow for populating a target from a database" on page 356
5. "Develop a message flow using a user-defined ESQL function" on page 357
6. "Develop a message flow using a user-defined Java procedure" on page 360

Develop a message flow that contains other nodes:

This is the first stage of the scenario to perform simple message enrichment. This topic demonstrates the following procedures:

- developing a message flow that contains other nodes (for example, a Filter node)
 - using mappings with conditions
 - developing corresponding message models, which use all main data types, and instance messages
1. From the Broker Application Development perspective, create the following resources:

- a message set project and a message set called MAPPING3_COMPLEX_messages, ensuring that the message set is namespace enabled with XML wire format
 - a message definition file called COMPLEX, which has a target namespace www.complex.net, with prefix comp
2. Create messages addev1, addev1s and addev1n with the following structures:

```

addev1
  bool      (xsd:boolean) local element
  bin       (xsd:hexBinary) local element
  dat       (xsd:dateTime) local element
  dec       (xsd:decimal) local element
  dur       (xsd:duration) local element
  flt       (xsd:float) local element
  int       (xsd:int) local element
  str       (xsd:string) local element

addev1s
  bin       (xsd:hexBinary) local element
  dat       (xsd:dateTime) local element
  dur       (xsd:duration) local element
  str       (xsd:string) local element

addev1n
  dec       (xsd:decimal) local element
  flt       (xsd:float) local element
  int       (xsd:int) local element

```

3. Create a message flow project called MAPPING3_COMPLEX_flows.
4. Create a message flow called addev1 which contains:

```

MQInput ->Filter -> Mapping -> Compute
                        \
                        \-> Mapping1-----/ \ --> RCD -> MQOutput

```

5. In the Filter node, set the following ESQL:

```

IF Body.bool THEN
    RETURN TRUE;
ELSE
    RETURN FALSE;
END IF;

```
6. In the Mapping node that is connected to the Filter true terminal (Mapping1), open the map and select addev1 as source and addev1s as target.
7. Wire the source to target as shown:

```

bin --- bin
dat --- dat
dur --- dur
str --- str

```
8. In the Spreadsheet pane, expand Properties and set the following values:

```

MessageType | 'addev1s'

```
9. Right-click the target dat and click **If**.
10. Replace the condition fn:true() with \$source/comp:addev1/str = 'dat'.
11. Set the value for dat to \$source/comp:addev1/dat + xs:duration("P3M").
12. Right-click the condition and click **Else**.
13. Right-click the target dur and click **If**.
14. Replace the condition fn:true() with \$source/comp:addev1/str = 'dur'.
15. Set the value for dur to \$source/comp:addev1/dur + xs:duration("P1Y").
16. Right-click the condition and click **Else**.
17. Open the map for the node that is connected to the false terminal of the Filter node (Mapping) and select addev1 as source and addev1n as target.

18. Wire the source to target as shown:

```
dec --- dec
flt --- flt
int --- int
```

19. In the Spreadsheet pane, expand Properties and set the following values:

```
MessageType | 'addev1n'
```

20. Set the ESQL in the Compute node to:

```
CALL CopyMessageHeaders();
SET OutputRoot.MRM.dec = InputBody.dec * 10;
SET OutputRoot.MRM.flt = InputBody.flt * 10;
SET OutputRoot.MRM.int = InputBody.int * 10;
```

21. In the ResetContentDescriptor node, set the Message Domain to XMLNS and select the **Reset Message Domain** check box.

22. Create three instance messages with the appropriate RFH2 headers:

```
<comp:addev1 xmlns:comp="http://www.complex.net">
<bool>1</bool>
<bin><![CDATA[010203]]></bin>
<dat>2005-05-06T00:00:00+00:00</dat>
<dec>19.34</dec>
<dur>P2Y4M</dur>
<flt>3.245E+2</flt>
<int>2104</int>
<str>dat</str>
</comp:addev1>

<comp:addev1 xmlns:comp="http://www.complex.net">
<bool>1</bool>
<bin><![CDATA[010203]]></bin>
<dat>2005-05-06T00:00:00+00:00</dat>
<dec>19.34</dec>
<dur>P2Y4M</dur>
<flt>3.245E+2</flt>
<int>2104</int>
<str>dur</str>
</comp:addev1>

<comp:addev1 xmlns:comp="http://www.complex.net">
<bool>0</bool>
<bin><![CDATA[010203]]></bin>
<dat>2005-05-06T00:00:00+00:00</dat>
<dec>19.34</dec>
<dur>P2Y4M</dur>
<flt>3.245E+2</flt>
<int>2104</int>
<str>dat</str>
</comp:addev1>
```

You have created the following resources:

- a message set called MAPPING3_COMPLEX_messages, which is populated with the messages addev1, addev1s and addev1n
- a message flow called addev1 in the project MAPPING3_COMPLEX_flows, which contains the mapping files addev1_Mapping.msgmap and addev1._Mapping1.msgmap
- files that contain instance messages for test

Now deploy the message set and message flow.

Deploy the message set and message flow:

This is the second stage of the scenario to perform simple message enrichment.

This topic demonstrates how to deploy the message set and message flow and run the instance messages through the broker.

1. Create a bar file called addev1.
2. Add the message set MAPPING3_COMPLEX_messages and the message flow addev1 to the bar file.
3. Deploy the bar file to the broker.
4. Put the instance documents on the input queue.

The output messages look like this:

```
<comp:addev1s xmlns:comp="http://www.complex.net">
<bin><![CDATA[010203]]></bin>
<dat>2005-08-06T00:00:00-01:00</dat>
<dur>P2Y4M</dur>
<str>dat</str>
</comp:addev1s>
```

Now go to “Develop a message flow to map target fields from multiple other fields.”

Develop a message flow to map target fields from multiple other fields:

Before you start

Perform the steps in the following topic:

1. “Develop a message flow that contains other nodes” on page 345

This is the third stage of the scenario to perform simple message enrichment. This topic demonstrates how to develop a message flow to map target fields from multiple other fields and also involves developing corresponding message models and instance documents.

1. In the COMPLEX message definition, in namespace www.complex.net, create a message called addev2, which has the following structure:

```
addev2
  firstname      (xsd:string) local element
  lastname       (xsd:string) local element
  branch         (xsd:string) local element
  accountno      (xsd:string) local element
  balance        (xsd:decimal) local element
  transvalue     local complex element, base type xsd:decimal
  transdir       (xsd:string) local attribute
```

2. In the message set MAPPING3_COMPLEX_messages, create a new message definition file called COMP2, which has the target namespace www.comp2.net, with prefix c2.
3. In the COMP2 message definition, create a message called addev2out, which has the structure:

```
addev2out
  accountdetails (xsd:string) local element
  transvalue     (xsd:decimal) local element
  balance        (xsd:decimal) local element
```

4. Create a message flow called addev2, which contains the following mapping: MQInput -> Mapping -> MQOutput.
5. Open the map and select addev2 as the source and addev2out as the target.
6. Wire the source to target as shown:

```
accountno --- accountdetails
balance --- balance
transvalue --- transvalue
```

7. In the Spreadsheet pane, expand Properties and set the following values:

MessageType | 'addev2out'

8. Set the accountdetails target to `fn:concat($source/comp:addev2/accountno, $source/comp:addev2/branch, $source/comp:addev2/lastname, $source/comp:addev2/firstname)`.
9. Right-click the target transvalue and click **If**.
10. Change the condition from `fn:true()` to `$source/comp:addev2/transvalue/@transdir = 'DEBIT'`.
11. Select transvalue and set its value to `$source/comp:addev2/transvalue * (-1)`.
12. Right-click the condition and click **Else**.
13. Right-click the target balance and click **If**.
14. Change the condition from `fn:true()` to `$source/comp:addev2/transvalue/@transdir = 'DEBIT'`.
15. Select balance and set its value to `$source/comp:addev2/balance - $source/comp:addev2/transvalue`.
16. Right-click the condition and click **Condition**.
17. Change the condition from `fn:true()` to `$source/comp:addev2/transvalue/@transdir = 'CREDIT'`.
18. Select balance following the second condition and set its Value to `$source/comp:addev2/balance + $source/comp:addev2/transvalue`.
19. Create two instance messages with the appropriate RFH2 headers:

```
<comp:addev2 xmlns:comp="http://www.complex.net">
<firstname>Brian</firstname>
<lastname>Benn</lastname>
<branch>52-84-02</branch>
<accountno>567432876543</accountno>
<balance>1543.56</balance>
<transvalue transdir="DEBIT">25.28</transvalue>
</comp:addev2>

<comp:addev2 xmlns:comp="http://www.complex.net">
<firstname>Brian</firstname>
<lastname>Benn</lastname>
<branch>52-84-02</branch>
<accountno>567432876543</accountno>
<balance>1543.56</balance>
<transvalue transdir="CREDIT">25.28</transvalue>
</comp:addev2>
```

You have created the following resources:

- a message called `addev2` in the message definition called `COMPLEX`
- a message called `addev2out` in the message definition called `COMP2`
- a message flow called `addev2`, which contains the mapping file `addev2_Mapping.msgmap`
- files that contain instance messages for test

Now deploy the message set and message flow

Deploy the message set and message flow:

This is the fourth stage of the scenario to perform simple message enrichment. This topic demonstrates how to deploy the message set and message flow and run the instance messages through the broker.

1. Create a bar file called `addev2`.
2. Add the message set `MAPPING3_COMPLEX_messages` and the message flow `addev2` to the bar file.

3. Deploy the bar file to the broker.
4. Put the instance documents on the input queue.

The output messages look like this:

```
<c2:addev2out xmlns:c2="http://www.comp2.net" xmlns:comp="http://www.complex.net">
<accountdetails>567432876543 52-84-02 Benn Brian</accountdetails>
<transvalue>-25.28</transvalue>
<balance>1518.28</balance>
</c2:addev2out>
```

Now go to “Develop a message flow and message model for mapping a complex nested, repeating message.”

Develop a message flow and message model for mapping a complex nested, repeating message:

Before you start

Perform the steps in the following topics:

1. “Develop a message flow that contains other nodes” on page 345
2. “Develop a message flow to map target fields from multiple other fields” on page 348

This is the fifth stage of the scenario to perform simple message enrichment. This topic demonstrates how to develop a message flow and message model for mapping a complex nested, repeating message. It also involves developing corresponding instance documents.

1. In the COMPLEX message definition, in namespace ‘www.complex.net’, create a message called addev3, which has the following structure:

```
addev3
  choice
    sstr                (xsd:string) local element
    intrep              (xsd:int) local element, minOcc=2, maxOcc=6
    dur                 (xsd:duration) local element
  choice
    comp1               local complex element
    dat1                (xsd:date) local element
    sval               (xsd:string) local element
    comp2               local complex element
    boo1               (xsd:boolean) local element
    dat2               (xsd:date) local element
    comprep             local complex element, minOcc=1, maxOcc=4
    int1               (xsd:int) local element
    decl               (xsd:decimal) local element
  bine1               (xsd:hexBinary) local element
  lelem               local complex element, base type xsd:string
  latt               (xsd:int) local attribute
  lcomp               local complex element
  head               (xsd:string) local element
  incomp              local complex element
  count               (xsd:int) local element
  comp:gcomp1         global complex element, minOcc=0, maxOcc=-1
  fstr                (xsd:string) local element
  multel              local complex element
  in1                 (xsd:boolean) local element
  in2                 (xsd:string) local element
  in3                 (xsd:float) local element
  footer              (xsd:string) local element
  repstr              (xsd:string) local element, minOcc=1, maxOcc=-1
```

2. Create a message flow called addev3, which contains the following mapping: MQInput > Mapping > MQOutput.
3. Open the map and select addev3 as the source and target.
4. Map each source element to its corresponding target element:

```
sstr --- sstr
intrep --- intrep
dur --- dur
dat1 --- dat1
sval --- sval
bool1 --- bool1
dat2 --- dat2
int1 --- int1
decl --- decl
binel --- binel
lelem --- lelem
latt --- latt
head --- head
count --- count
fstr --- fstr
multel --- multel
footer --- footer
repstr --- repstr
```

5. In the Spreadsheet pane, for the first condition, change fn:true() to fn:exists(\$source/comp:addev3/sstr).
6. For the second condition, change fn:true() to fn:exists(\$source/comp:addev3/intrep).
7. For the third condition, change fn:true() to fn:exists(\$source/comp:addev3/dur).
8. For the first complex choice condition, change fn:true() to fn:exists(\$source/comp:addev3/comp1).
9. For the second complex choice condition, change fn:true() to fn:exists(\$source/comp:addev3/comp2).
10. For the third complex choice condition, change fn:true() to fn:exists(\$source/comp:addev3/comp3).
11. Create the following instance messages, with appropriate RFH2 headers:

```
<comp:addev3 xmlns:comp="http://www.complex.net">
<sstr>first</sstr>
<comp1>
<dat1>2005-06-24</dat1>
<sval>date value</sval>
</comp1>
<binel><![CDATA[3132333435]]></binel>
<lelem latt="24">twenty four</lelem>
</comp>
<head>nesting start</head>
<incomp>
<count>3</count>
<comp:gcomp1>
<fstr>first</fstr>
<multel>
<in1>1</in1>
<in2>C</in2>
<in3>2.45E+1</in3>
</multel>
</comp:gcomp1>
<comp:gcomp1>
<fstr>second</fstr>
<multel>
<in1>1</in1>
<in2>D</in2>
```

```

<in3>7.625E+3</in3>
</multel>
</comp:gcompel>
<comp:gcompel>
<fstr>third</fstr>
<multel>
<in1>0</in1>
<in2>C</in2>
<in3>4.9E+0</in3>
</multel>
</comp:gcompel>
</incomp>
<footer>nesting end</footer>
</lcomp>
<repstr>abc</repstr>
<repstr>def</repstr>
<repstr>ghi</repstr>
<repstr>jkl</repstr>
<repstr>mno</repstr>
</comp:addev3>

<comp:addev3 xmlns:comp="http://www.complex.net">
<intrep>45</intrep>
<intrep>12</intrep>
<intrep>920</intrep>
<comp2>
<bool1>1</bool1>
<dat2>2005-06-24</dat2>
</comp2>
<binel><![CDATA[3132333435]]></binel>
<lelem latt="24">twenty four</lelem>
<lcomp>
<head>nesting start</head>
<incomp>
<count>5</count>
<comp:gcompel>
<fstr>first</fstr>
<multel>
<in1>1</in1>
<in2>C</in2>
<in3>2.45E+1</in3>
</multel>
</comp:gcompel>
<comp:gcompel>
<fstr>second</fstr>
<multel>
<in1>1</in1>
<in2>D</in2>
<in3>7.625E+3</in3>
</multel>
</comp:gcompel>
<comp:gcompel>
<fstr>third</fstr>
<multel>
<in1>0</in1>
<in2>C</in2>
<in3>4.9E+0</in3>
</multel>
</comp:gcompel>
<comp:gcompel>
<fstr>fourth</fstr>
<multel>
<in1>1</in1>
<in2>F</in2>
<in3>2.98E+1</in3>
</multel>
</comp:gcompel>
<comp:gcompel>

```



```

<fstr>fifth</fstr>
<multel>
<in1>0</in1>
<in2>D</in2>
<in3>8.57E-2</in3>
</multel>
</comp:gcomp1>
</incomp>
<footer>nesting end</footer>
</lcomp>
<repstr>abc</repstr>
</comp:addev3>

<comp:addev3 xmlns:comp="http://www.complex.net">
<dur>P2Y2M</dur>
<comp3>
<int1>6</int1>
<dec1>2821.54</dec1>
</comp3>
<comp3>
<int1>41</int1>
<dec1>0.02</dec1>
</comp3>
<binel><![CDATA[3132333435]]></binel>
<lelem latt="24">twenty four</lelem>
<lcomp>
<head>nesting start</head>
<incomp>
<count>0</count>
</incomp>
<footer>nesting end</footer>
</lcomp>
<repstr>abc</repstr>
<repstr>def</repstr>
<repstr>ghi</repstr>
<repstr>jkl</repstr>
<repstr>mno</repstr>
<repstr>pqr</repstr>
<repstr>stu</repstr>
<repstr>vwx</repstr>
</comp:addev3>

```

You have created the following resources:

- a message called addev3 in the message definition COMPLEX
- a message flow called addev3, which contains the mapping file addev3_Mapping.msgmap
- files that contain instance messages for test

Now deploy the message set and message flow.

Deploy the message set and message flow:

This is the sixth stage of the scenario to perform simple message enrichment. This topic demonstrates how to deploy the message set and message flow and run the instance messages through the broker.

1. Create a bar file called addev3.
2. Add the message set MAPPING3_COMPLEX_messages and the message flow addev3 to the bar file.
3. Deploy the bar file to the broker.
4. Put the instance documents on the input queue.

The output messages look like this:

```

<comp:addev3 xmlns:comp="http://www.complex.net">
<sstr>first</sstr>
<comp1>
<dat1>2005-06-24</dat1>
<sval>date value</sval>
</comp1>
<binel><![CDATA[3132333435]]></binel>
<lelem latt="24">twenty four</lelem>
<lcomp>
<head>nesting start</head>
<incomp>
<count>3</count>
<comp:gcompel>
<fstr>first</fstr>
<multel>
<in1>1</in1>
<in2>C</in2>
<in3>2.45E+1</in3>
</multel>
</comp:gcompel>
<comp:gcompel>
<fstr>second</fstr>
<multel>
<in1>1</in1>
<in2>D</in2>
<in3>7.625E+3</in3>
</multel>
</comp:gcompel>
<comp:gcompel>
<fstr>third</fstr>
<multel>
<in1>0</in1>
<in2>C</in2>
<in3>4.9E+0</in3>
</multel>
</comp:gcompel>
</incomp>
<footer>nesting end</footer>
</lcomp>
<repstr>abc</repstr>
<repstr>def</repstr>
<repstr>ghi</repstr>
<repstr>jkl</repstr>
<repstr>mno</repstr>
</comp:addev3>

<comp:addev3 xmlns:comp="http://www.complex.net">
<intrep>45</intrep>
<intrep>12</intrep>
<intrep>920</intrep>
<comp2>
<bool1>1</bool1>
<dat2>2005-06-24</dat2>
</comp2>
<binel><![CDATA[3132333435]]></binel>
<lelem latt="24">twenty four</lelem>
<lcomp>
<head>nesting start</head>
<incomp>
<count>5</count>
<comp:gcompel>
<fstr>first</fstr>
<multel>
<in1>1</in1>
<in2>C</in2>
<in3>2.45E+1</in3>
</multel>
</comp:gcompel>

```

```

<comp:gcomp1>
<fstr>second</fstr>
<multel>
<in1>1</in1>
<in2>D</in2>
<in3>7.625E+3</in3>
</multel>
</comp:gcomp1>
<comp:gcomp1>
<fstr>third</fstr>
<multel>
<in1>0</in1>
<in2>C</in2>
<in3>4.9E+0</in3>
</multel>
</comp:gcomp1>
<comp:gcomp1>
<fstr>fourth</fstr>
<multel>
<in1>1</in1>
<in2>F</in2>
<in3>2.98E+1</in3>
</multel>
</comp:gcomp1>
<comp:gcomp1>
<fstr>fifth</fstr>
<multel>
<in1>0</in1>
<in2>D</in2>
<in3>8.57E-2</in3>
</multel>
</comp:gcomp1>
</incomp>
<footer>nesting end</footer>
</lcomp>
<repstr>abc</repstr>
</comp:addev3>

<comp:addev3 xmlns:comp="http://www.complex.net">
<dur>P2Y2M</dur>
<comp3>
<int1>6</int1>
<dec1>2821.54</dec1>
</comp3>
<comp3>
<int1>41</int1>
<dec1>0.02</dec1>
</comp3>
<binel><![CDATA[3132333435]]></binel>
<lelem latt="24">twenty four</lelem>
<lcomp>
<head>nesting start</head>
<incomp>
<count>0</count>
</incomp>
<footer>nesting end</footer>
</lcomp>
<repstr>abc</repstr>
<repstr>def</repstr>
<repstr>ghi</repstr>
<repstr>jkl</repstr>
<repstr>mno</repstr>
<repstr>pqr</repstr>
<repstr>stu</repstr>
<repstr>vwx</repstr>
</comp:addev3>

```

Now go to “Develop a message flow for populating a target from a database.”

Develop a message flow for populating a target from a database:

Before you start

Perform the steps in the following topics:

1. “Develop a message flow that contains other nodes” on page 345
2. “Develop a message flow to map target fields from multiple other fields” on page 348
3. “Develop a message flow and message model for mapping a complex nested, repeating message” on page 350

This is the seventh stage of the scenario to perform simple message enrichment. This topic demonstrates how to develop a message flow for populating a target from a database. It also involves developing a corresponding message model and instance documents.

1. Create a database called MAPDB and create a table called TRANSACTION, which has the following columns:

ACCOUNT	VARCHAR(12)
TDATE	DATE
VALUE	DECIMAL(8,2)

2. Populate the database with the values shown:

```
'12345678901', '2005-04-25', -14.25
'12345678901', '2005-04-25', 100.00
'12345678901', '2005-05-15', 2891.30
'12345678901', '2005-06-11', -215.28
```

3. Create a Windows ODBC Data Source Name for the database and then register the database with the Configuration Manager by clicking **File** → **New** → **RDB Definitions File**.
4. In the COMPLEX message definition, in namespace `www.complex.net`, create a message called `addev4in`, which has the following structure:

```
addev4in
  account          (xsd:string) local element
  tdate            (xsd:date) local element
```

5. In the COMP2 message definition, in namespace `www.comp2.net`, create a message called `addev4out`, which has the following structure:

```
addev4out
  account          (xsd:string) local element
  tdate            (xsd:date) local element
  value            (xsd:decimal) local element, min0cc=0, max0cc=-1
```

6. Create a message flow called `addev4`, which contains the following mapping: `MQInput` > `Mapping` > `MQOutput`.
7. Open the map and select `addev4in` as the source and `addev4out` as the target.
8. Map the input to outputs as shown:

```
account --- account
tdate --- tdate
```

9. In the Spreadsheet pane, right-click the target value and click **Select Data Source**.
10. Select MAPDB from the dialog box and click **Finish**.
11. In the top pane, expand the MAPDB tree and wire the values as shown:

```
VALUE --- value
```

12. In the Spreadsheet pane, select the target \$db:select and change fn:true() to: \$db:select.MAPDB.SCHEMA.TRANSACTION.ACCOUNT=\$source/comp:addev4in/account and \$db:select.MAPDB.SCHEMA.TRANSACTION.TDATE=\$source/comp:addev4in/tdate
13. Expand the Properties tree and set the following values:
 MessageType | 'addev4out'
14. Set the data source property for the Mapping node to MAPDB.
15. Create the following instance messages with appropriate RFH2 headers:

```
<comp:addev4in xmlns:comp="http://www.complex.net">
<account>12345678901</account>
<tdate>2005-05-15</tdate>
</comp:addev4in>

<comp:addev4in xmlns:comp="http://www.complex.net">
<account>12345678901</account>
<tdate>2005-04-25</tdate>
</comp:addev4in>
```

You have created the following resources:

- a message called addev4in in a message definition called COMPLEX
- a message called addev4out in a message definition called COMP
- a message flow called addev4, which contains the mapping file addev4_Mapping.msgmap
- files that contain instance messages

Now deploy the message set and message flow.

Deploy the message set and message flow:

This is the eighth stage of the scenario to perform simple message enrichment. This topic demonstrates how to deploy the message set and message flow and run the instance messages through the broker.

1. Create a bar file called addev4.
2. Add the message set MAPPING3_COMPLEX_messages and the message flow addev4 to the bar file.
3. Deploy the bar file to the broker.
4. Put the instance documents on the input queue.

The output messages look like this:

```
<c2:addev4out xmlns:c2="http://www.comp2.net" xmlns:comp="http://www.complex.net" >
<account>12345678901</account>
<tdate>2005-05-15</tdate>
<value>2891.3</value>
</c2:addev4out>
```

Now go to “Develop a message flow using a user-defined ESQL function.”

Develop a message flow using a user-defined ESQL function:

Before you start

Perform the steps in the following topics:

1. “Develop a message flow that contains other nodes” on page 345
2. “Develop a message flow to map target fields from multiple other fields” on page 348

3. “Develop a message flow and message model for mapping a complex nested, repeating message” on page 350
4. “Develop a message flow for populating a target from a database” on page 356

This is the ninth stage of the scenario to perform simple message enrichment. This topic demonstrates how to develop a message flow using a user-defined ESQL function. It also involves developing corresponding message models and instance documents.

1. In the COMPLEX message definition, in namespace `www.complex.net`, create messages called `addev5in` and `addev5out`, which have the following structures:

```

addev5in
  value1          (xsd:decimal) local element
  operator        (xsd:string) local element
  value2          (xsd:decimal) local element
  rate            (xsd:decimal) local element

addev5out
  grossvalue      (xsd:decimal) local element
  netvalue        (xsd:decimal) local element

```

2. Create a message flow called `addev5`, which contains the following mapping: `MQInput > Mapping > MQOutput`.
3. Open the map and select `addev5in` as the source and `addev5out` as the target.
4. In the `MAPPING3_COMPLEX_flows` project, create an ESQL file called `addev5` and put these functions in it:

```

CREATE FUNCTION calcGrossvalue(IN value1 DECIMAL, IN operator CHAR,
  IN value2 DECIMAL) RETURNS DECIMAL
  BEGIN
    DECLARE outval DECIMAL;
    CASE operator
      WHEN 'PLUS' THEN
        SET outval = value1 + value2;
      WHEN 'MINUS' THEN
        SET outval = value1 - value2;
      WHEN 'MULTIPLY' THEN
        SET outval = value1 * value2;
      WHEN 'DIVIDE' THEN
        SET outval = value1 / value2;
      ELSE
        THROW USER EXCEPTION MESSAGE 2949 VALUES('Invalid Operator', operator);
        SET outval = -999999;
    END CASE;
    RETURN outval;
  END;

CREATE FUNCTION calcNetvalue(IN value1 DECIMAL, IN operator CHAR, IN value2 DECIMAL,
  IN rate DECIMAL) RETURNS DECIMAL
  BEGIN
    DECLARE grossvalue DECIMAL;
    SET grossvalue=calcGrossvalue(value1, operator, value2);
    RETURN (grossvalue * rate );
  END;

```

5. In the Message Mapping editor Spreadsheet pane, expand the message and select `grossvalue`.
6. In the Expression pane, enter the expression:


```

esql:calcGrossvalue($source/comp:addev5in/value1,
  $source/comp:addev5in/operator,
  $source/comp:addev5in/value2)

```
7. Select the target `netvalue`, and in the Expression pane, enter the following expression:

```
esql:calcNetvalue($source/comp:addev5in/value1,
$source/comp:addev5in/operator,
$source/comp:addev5in/value2,
$source/comp:addev5in/rate)
```

- Expand the Properties tree and set the following values:

```
MessageType      |      'addev5out'
```

- Create the following instance messages, with appropriate RFH2 headers:

```
<comp:addev5in xmlns:comp="http://www.complex.net">
<value1>125.32</value1>
<operator>PLUS</operator>
<value2>25.86</value2>
<rate>0.60</rate>
</comp:addev5in>

<comp:addev5in xmlns:comp="http://www.complex.net">
<value1>118.00</value1>
<operator>MINUS</operator>
<value2>245.01</value2>
<rate>0.30</rate>
</comp:addev5in>

<comp:addev5in xmlns:comp="http://www.complex.net">
<value1>254.02</value1>
<operator>MULTIPLY</operator>
<value2>3.21</value2>
<rate>0.75</rate>
</comp:addev5in>

<comp:addev5in xmlns:comp="http://www.complex.net">
<value1>1456.33</value1>
<operator>DIVIDE</operator>
<value2>18.58</value2>
<rate>0.92</rate>
</comp:addev5in>

<comp:addev5in xmlns:comp="http://www.complex.net">
<value1>254.02</value1>
<operator>MOD</operator>
<value2>3.21</value2>
<rate>0.75</rate>
</comp:addev5in>
```

You have created the following resources:

- messages called addev5in and addev5out in a message definition called COMPLEX
- a message flow called addev5, which contains the mapping file addev5_Mapping.msgmap and ESQL file addev5.esql
- files containing instance messages

Now deploy the message set and message flow.

Deploy the message set and message flow:

This is the tenth stage of the scenario to perform simple message enrichment. This topic demonstrates how to deploy the message set and message flow and run the instance messages through the broker.

1. Create a bar file called addev5.
2. Add the message set MAPPING3_COMPLEX_messages and the message flow addev5 to the bar file.
3. Deploy the bar file to the broker.
4. Put the instance documents on the input queue.

The output messages look like this:

```

<comp:addev5out xmlns:comp="http://www.complex.net">
<grossvalue>151.18</grossvalue>
<netvalue>90.708</netvalue>
</comp:addev5out>

<comp:addev5out xmlns:comp="http://www.complex.net">
<grossvalue>-127.01</grossvalue>
<netvalue>-38.103</netvalue>
</comp:addev5out>

<comp:addev5out xmlns:comp="http://www.complex.net">
<grossvalue>815.4042</grossvalue>
<netvalue>611.55315</netvalue>
</comp:addev5out>

<comp:addev5out xmlns:comp="http://www.complex.net">
<grossvalue>78.38159311087190527448869752421959</grossvalue>
<netvalue>72.11106566200215285252960172228202</netvalue>
</comp:addev5out>

```

If there is no message output look for an entry in the event log like this:

```

BIP2949 ( BRK.default ) A user generated ESQL exception has been thrown. The additional
information provided with this exception is: 'Invalid Operator' 'MOD'
'addev5.Mapping.ComIbmCompute' '%5' '%6' '%7' '%8' '%9' '%10' '%11'

```

This exception was thrown by a THROW EXCEPTION statement. This is the normal behavior of the THROW statement; this is a user-generated exception, so the user action is determined by the message flow and the type of exception that is thrown.

Now go to “Develop a message flow using a user-defined Java procedure.”

Develop a message flow using a user-defined Java procedure:

Before you start

Perform the steps in the following topics:

1. “Develop a message flow that contains other nodes” on page 345
2. “Develop a message flow to map target fields from multiple other fields” on page 348
3. “Develop a message flow and message model for mapping a complex nested, repeating message” on page 350
4. “Develop a message flow for populating a target from a database” on page 356
5. “Develop a message flow using a user-defined ESQL function” on page 357

This is the eleventh stage of the scenario to perform simple message enrichment. This topic demonstrates how to develop a message flow using a user-defined Java procedure. It also involves developing corresponding message models and instance documents.

1. In the COMPLEX message definition, in namespace www.complex.net, create messages called addev6in and addev6out, which have the following structures:

```

addev6in
  hexdata          (xsd:hexBinary) local element
addev6out
  decval          (xsd:decimal) local element
  fltval         (xsd:float) local element
  intval         (xsd:int) local element

```


2. Create a message flow called addev6, which contains the following mapping: MQInput > Mapping > MQOutput.
3. Open the map and select addev6in as the source and addev6out as the target.
4. In the MAPPING3_COMPLEX_flows project, create an ESQ file called addev6 and put these functions in it:

```
CREATE PROCEDURE decFromBinary( IN hexval BLOB )
  RETURNS DECIMAL
  LANGUAGE JAVA
  EXTERNAL NAME "addev6.decFromBinary";
CREATE PROCEDURE fltFromBinary( IN hexval BLOB )
  RETURNS DECIMAL
  LANGUAGE JAVA
  EXTERNAL NAME "addev6.fltFromBinary";
CREATE PROCEDURE intFromBinary( IN hexval BLOB )
  RETURNS DECIMAL
  LANGUAGE JAVA
  EXTERNAL NAME "addev6.intFromBinary";
```

5. Create a java source file called addev6.java, which has the following contents:

```
import java.lang.*;
import java.math.*;

public class addev6 {
    //
    // Return decimal element from binary string
    //
    public static BigDecimal decFromBinary( byte[] hexval) {
        // Look for element named decval
        String search = "decval";
        String sval = findElement(hexval ,search );
        // Convert the value to decimal type
        BigDecimal numval = new BigDecimal(sval);
        return numval;
    }
    //
    // Return float element from binary string
    //
    public static Double fltFromBinary( byte[] hexval) {
        // Look for element named fltval
        String search = "fltval";
        String sval = findElement(hexval ,search );
        // Convert the value to float type
        Double numval = new Double(sval);
        return numval;
    }
    //
    // Return integer element from binary string
    //
    public static Long intFromBinary( byte[] hexval) {
        // Look for element named intval
        String search = "intval";
        String sval = findElement(hexval ,search );
        // Convert the value to integer type
        Long numval = new Long(sval);
        return numval;
    }
    //
    // Locate the named element and its value in the binary data
    //
    private static String findElement( byte[] hexval, String search ) {
        // Convert bytes to string
        String hexstr = new String(hexval);
        // Fixed length label/value pairs (length=14)
        int nvals = hexstr.length() / 14;
        String numval = "";
```

```

String[] label = new String[nvals];
String[] value = new String[nvals];
// Loop over number of label/value pairs
for ( int i=0; i < nvals; i ++ ) {
    // get start position
    int st = i * 14;
    // label is length 6
    int endl = st + 6;
    // value is length 8
    int endv = endl + 8;
    // extract label and value from string
    label[i] = hexstr.substring( st, endl);
    value[i] = hexstr.substring( (endl+1), endv);
    // Check whether the current pair has the label requested
    if ( label[i].compareTo( search) == 0 ) {
        // trim padding from the value
        numval = value[i].trim();
    }
}
return numval;
}
}

```

6. Compile the java code and add the location of the class file to the system classpath. You might need to restart Windows if you edit the CLASSPATH.
7. In the Spreadsheet pane of the Message Mapping editor, expand the target message and set the target decval to the value esql:decFromBinary(\$source/comp:addev6in/bval).
8. Set the target fltval to esql:fltFromBinary(\$source/comp:addev6in/bval).
9. Set the target intval to esql:intFromBinary(\$source/comp:addev6in/bval).
10. Expand the Properties target and set the values shown:

MessageType		'addev6out
-------------	--	------------
11. Create the following instance message, with appropriate RFH2 headers:


```

<comp:addev6in xmlns:comp="http://www.complex.net">
  <bval>
    <![CDATA[64656376616c20202031342e3238666c7476616c
2020312e34452b32696e7476616c2020202020313230]]>
  </bval>
</comp:addev6in>

```

You have created the following resources:

- messages called addev6in and addev6out in a message definition called COMPLEX
- a message flow called addev6, which contains the mapping file addev6_Mapping.msgmap and ESQl file addev6.esql
- a Java source file called addev6.java and a compiled class file called addev6.class in a place where the system CLASSPATH can find it
- files that contain instance messages

Now deploy the message set and message flow.

Deploy the message set and message flow:

This is the final stage of the scenario to perform simple message enrichment. This topic demonstrates how to deploy the message set and message flow and run the instance message through the broker.

1. Create a bar file called addev6.
2. Add the message set MAPPING3_COMPLEX_messages and the message flow addev6 to the bar file.

3. Deploy the bar file to the broker.
4. Put the instance documents on the input queue.

The output message looks like this:

```
<comp:addev6out xmlns:comp="http://www.complex.net">
<decval>14.28</decval>
<fltval>1.4E+2</fltval>
<intval>120</intval>
</comp:addev6out>
```

You have completed this scenario.

Scenario E: Resolving a choice with alternative message data

Before you start:

1. Create the appropriate message model, either by using the tooling or by importing the message structure files (for example, C header or XML Schema Definition files).
2. Create a message flow that has the following structure:
MQInput > Mapping node > MQOutput

This scenario demonstrates how to resolve a choice with alternative message data. The message model used in this example is:

```
chsmess (message)
  head (xsd:string)
  choice (group)
    str1 (xsd:string)
    int1 (xsd:int)
    dur1 (xsd:duration)
  footer (xsd:string)
```

1. Switch to the Broker Application Development perspective.
2. Right-click the Mapping node and click **Open Map**.
3. Accept the default project and name, and click **Next**.
4. Accept the default usage and click **Next**.
5. Clear the **Based on records in a database** check box and click **Next**.
6. Select the source message chsmess and the target message chsmess, and click **Finish**.
7. In the Connection pane, open the source and target trees by clicking on the addition (+) icons.
8. Open the chsmess tree in the Source and Target panes in the same way.
9. In both Source and Target panes, click the addition (+) icon adjacent to the choice group.
10. Click head in the Message Mapping editor Source pane and drag it onto head in the Target pane. A line joins them.
11. Repeat Step 10 for each corresponding element (str1, int1, dur1, and footer.)
12. In the Map Script | Value table, open the tree by clicking the \$target + box.
13. Open the chsmess tree, then open the if. A set of condition elements appears.
14. Open each condition. One condition exists for each choice. Each condition has the function fn:true().
15. Click the first function (for example, for str1) and change it in the Edit pane to: \$source/chsmess/head='str1. If the input element head has a value str1, the assignment str1 <- \$source/chsmess/str1 takes place.

16. Click the second function (for example, for int1) and change it in the Expression editor to: `$source/chsmess/head='int1'`. If the input element head has a value int1, the assignment `int1 <- $source/chsmess/int1` takes place.
17. Click the third function (for example, for dur1) and change it in the Expression editor to: `$source/chsmess/head='dur1'`. If the input element head has a value dur1, the assignment `dur1 <- $source/chsmess/dur1` takes place.
18. Save the mapping by clicking **File** → **Save**.

You have completed this scenario. The message model contains a choice that has been resolved using other data in the instance message.

Scenario F: Updating the value of a message element

Before you start:

1. Create the appropriate message model, either by using the tooling or by importing the message structure files (for example, C header or XML Schema Definition files).
2. Create a message flow that has the following structure:
MQInput > Mapping node > MQOutput

This scenario demonstrates how to update the value of a message element. The message model used in this example is:

```
simple (message)
  int (xsd:int)
  str (xsd:str)
```

1. Switch to the Broker Application Development perspective.
2. Right-click the Mapping node and click **Open Map**.
3. Accept the default project and name and click **Next**.
4. Accept the default usage and click **Next**.
5. Clear the **Based on records in a database** check box and click **Next**.
6. Select the source message simple and the target message simple and click **Finish**.
7. In the connection pane, open the source and target trees by clicking the addition (+) icons.
8. Open the simple trees on both sides in the same way.
9. Select int in the Message Mapping editor Source pane, and drag it onto int in the Target pane. A line joins them.
10. Select str in the Message Mapping editor Source pane and drag it onto str in the Target pane. A line joins them.
11. In the Map Script | Value table, open the tree by clicking the \$target + box
12. Open the simple tree in the same way; both int and str have values (for example, int `$source/simple/int` str `$source/simple/str`).
13. Select the value for int. The value appears in the Expression Editing pane.
14. Edit the value so that it is: `$source/simple/int + 1` and press **Enter**. The value in the table is updated (this increments the input value).
15. Select the value for str and edit it so that it is: `esql:upper($source/simple/str)`, and press **Enter**. The value in the table is updated (this converts the input value to upper case).
16. Save the mapping by clicking **File** > **Save**.

You have completed this scenario. The input and output messages have the same structure and format, but the element values have been modified.

Defining promoted properties

When you create a message flow, you can promote properties from individual nodes within that message flow to the message flow level. Properties promoted in this way override the property values that you have set for the individual nodes. You can perform the following tasks related to promoting properties:

- “Promoting a property”
- “Renaming a promoted property” on page 369
- “Removing a promoted property” on page 370
- “Converging multiple properties” on page 371

Some of the properties that you can promote to the message flow are also configurable; you can modify them when you deploy the broker archive file in which you have stored the message flow to each broker. If you set values for configurable properties when you deploy a broker archive file, the values that you set override values set in the individual nodes, and those that you have promoted.

Promoting a property

You can promote a node property to the message flow level to simplify the maintenance of the message flow and its nodes, and to provide common values for multiple nodes within the flow by converging promoted properties.

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow” on page 126

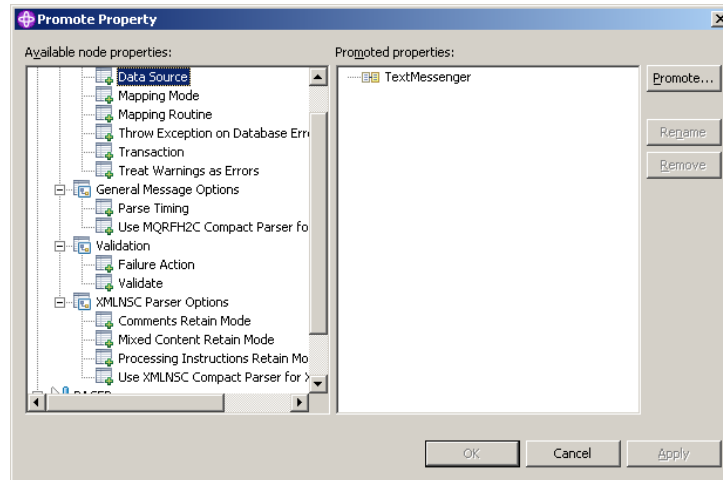
To promote message flow node properties to the message flow level:

1. Switch to the Broker Application Development perspective.
2. Open the message flow for which you want to promote properties by double-clicking the message flow in the Navigator view. You can also open the message flow by right-clicking it in the Navigator view and clicking **Open**. The message flow contents are displayed in the editor view.

If this is the first message flow that you have opened, the message flow control window and the list of available built-in message flow nodes are also displayed, to the left of the editor view.

3. In the editor view, right-click the symbol of the message flow node whose properties you want to promote.
4. Select **Promote Property**.

The Promote Property dialog is displayed.



The left side of the dialog lists all available properties for all the nodes within the message flow. The properties for the node that you highlighted are expanded. You can access the properties for all the nodes in the open message flow from this dialog, regardless of the node that you selected when you first opened the dialog, by expanding the properties for all the other nodes in the flow (these are initially collapsed).

The right side of the dialog lists the name of the open message flow and all the properties that are currently promoted to the message flow. If you have not yet promoted any properties, only the message flow name as the root of the promoted property tree is displayed on the right. In the image shown the message flow contains no promoted properties so only the name of the message flow is displayed.

The majority of message flow node properties are available for promotion, but you cannot promote the following properties:

- The properties that name Mapping modules.
 - A property group, but you can promote an individual property.
 - A property that you cannot edit (for example, the *Fix* property in the Validate group of properties for the MQInput node).
 - The description properties (Short Description and Long Description).
5. Select the property that you want to promote to the message flow. The list on the left initially shows the expanded list of all available properties for the selected node. If you have already promoted properties from this node, they do not appear on the left, but on the right.

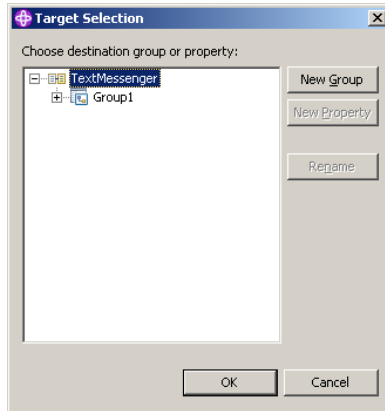
The list on the left also includes the other nodes in the open message flow. You can expand the properties listed under each node and work with all these properties at the same time. You do not have to close the dialog and select another node from the editor view to continue promoting properties.

You can select multiple properties to promote by selecting a property, holding down Ctrl, and selecting one or more other properties.

If you have you selected multiple properties to promote, all the properties you have selected must be available for promotion. If one or more of the selected properties is not available for promotion, the entire selection becomes unavailable for promotion, and the **Promote** button in the right-hand pane is grayed out.

6. Click the **Promote** button to promote the property or properties

Clicking the Promote button invokes the Target Selection dialog:



The Target Selection dialog displays only the valid targets for the promotion of the previously selected property or properties and allows you to create a new target for the promotion, such as to a new group or to a new property.

7. In the Target Selection dialog, select the destination group or property for the property or properties you want to promote. You can group together related properties from the same or different nodes in the message flow by dropping the selected property or properties onto a group or property that already exists. Alternatively, you can click **New Group** or **New Property** to create a new target for the promotion. You can rename groups and properties by selecting them and clicking **Rename**, or by double-clicking on the group or property.
8. Click **OK** to confirm your selections.

Note: If you create a new group or property using the Target Selection dialog, the changes persist even if you select **Cancel** in the dialog. When the dialog closes, groups or properties that you have created using the Target Selection dialog will appear in the Promote properties dialog.

9. When you have selected the properties that you want to promote to the message flow, click **OK**. Your updates are committed, and the Promoted Property dialog is closed. If you click **Apply**, this commits the changes but leaves the dialog open.

When you have promoted a property, you can no longer make any changes to that property through the node properties dialog. You can only update its value at the message flow level.

Note: You can also promote properties from the Promote property dialog by dragging the selected property or properties from the left-hand pane of the Promote Property dialog to the right-hand pane:

1. Select the property you want to promote. You can select multiple properties to promote by selecting a property, holding down Ctrl, and selecting one or more other properties.
2. You can drop the selected property or properties into the right-hand pane using the following methods:
 - a. Drop the selected property or properties in an empty space.

A new group is automatically created for the message flow, and the property is placed within it, with the original name of the property and the name of the message flow node from which it came displayed beneath the property entry.

The name of the first group created defaults to Group1. If a group called Group1 already exists, the group is given the name Group2,

and so on. You can rename the group by double-clicking it and entering new text or by selecting the group in the Promoted properties pane and clicking **Rename**.

Note:

When you create a new promoted property, the name that you enter is the name by which the property is known within the system, and must meet certain Java and XML naming restrictions. These are enforced by the dialog, and a message is displayed if you enter a name that includes a non-valid character. For example, you cannot include a space or the double quote symbol.

If you are developing a message flow within a user-defined project that will be delivered as an Eclipse plug-in, you can add translation for the promoted properties that you have added. Translated names can contain characters, such as space, that are restricted for system names. The option to provide translated strings for promoted properties is not available if you are working with a message flow within a message flow project.

- b. Drop the selected property or properties onto a group that already exists, to group together related properties from the same or different nodes in the message flow.

For example, you might want to group all promoted properties that relate to database interactions. You can change the groups that promoted properties belong to at any time, by selecting a property in the Promoted properties pane and dragging it onto a different group.

- c. Drop the selected property or properties onto a property that already exists, to converge related properties from the same or different nodes in the message flow.

For example, you might want to create a single promoted property that overrides the property on each node that defines a data source.

For more information on converging properties, see “Converging multiple properties” on page 371.

The message flow node properties are now promoted to the message flow. To confirm this, right-click the message flow in the Navigator view, or right-click the editor view, and select **Properties**.

The Properties dialog of the message flow is displayed, showing the message flow node properties that you have promoted, organized in the groups that you have created. If you now set a value for one of these properties, that value appears as the default value for the property whenever the message flow is itself included in other message flows.

When you have promoted a property, you can no longer make any changes to that property through the node properties dialog. You can only update its value at the message flow level.

When you select an embedded message flow within another message flow (a subflow) and view its properties, you see the promoted property values. If you look inside the embedded flow (that is, if you select **Open Subflow**), you see the

original values for the properties. The value of a promoted property does not replace the original property, but it takes precedence when you deploy the message flow.

Promoting mandatory properties

If you promote a property that is mandatory (that is, an asterisk appears beside the name in the properties dialog of the message flow node), the mandatory characteristic of the property is preserved. When a mandatory property is promoted, its value does not need to be set at the node-level. If the flow containing the mandatory promoted property is included as a subflow within another flow, then the property has to be filled in for the subflow node.

Promoting properties through a hierarchy of message flows

You can repeat the process of promoting message flow node properties through several levels of message flow. You can promote properties from any level in the hierarchy to the next level above, and so on through the hierarchy to the top level. The value of a property is propagated from the highest point in the hierarchy at which it is set down to the original message flow node when the message flow is deployed to a broker. The value of that property on the original message flow node is overridden.

Renaming a promoted property

If you have promoted a property from the node to the message flow level, it is initially assigned the same name that it has at the node level. You can rename the property to have a more meaningful name in the context of the message flow.

Before you start

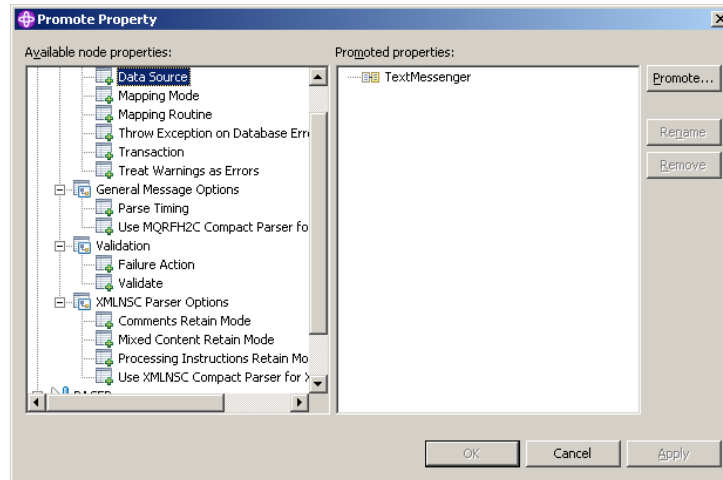
To complete this task, you must have completed the following task:

- “Promoting a property” on page 365

To rename a promoted property :

1. Switch to the Broker Application Development perspective.
2. Open the message flow for which you want to promote properties by double-clicking the message flow in the Navigator view. You can also open the message flow by right-clicking it in the Navigator view and clicking **Open**. The message flow contents are displayed in the editor view.
If this is the first message flow that you have opened, the message flow control window and the list of available built-in message flow nodes are also displayed, to the left of the editor view.
3. In the editor view, right-click the symbol of the message flow node whose properties you want to promote.
4. Select **Promote Property**.

The Promote Property dialog is displayed.



5. Promoted properties are shown in the Promoted properties pane on the right of the Promote property dialog. Double-click the promoted property in the list of properties that are currently promoted to the message flow level, or select the property you want to rename and click **Rename**. The name is highlighted, and you can edit it. Modify the existing text or enter new text to give the property a new name, and press Enter.
6. Click **Apply** to commit this change without closing the Property Promotion dialog. Click **OK** to complete your updates and close the dialog.

Removing a promoted property

If you have promoted a property from the node to the message flow level, you can remove (delete) it if you no longer want to specify its value at the message flow level. The property reverts to the value that you specified at the node level. If you remove a promoted property that is a mandatory property, ensure that you have set a value at the node level. If you have not, you cannot successfully deploy a broker archive file that includes this message flow.

Before you start

To complete this task, you must have completed the following task:

- “Promoting a property” on page 365

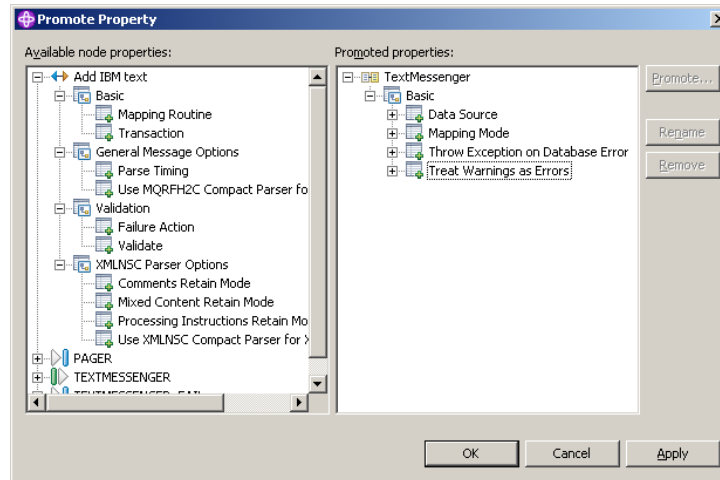
If you have promoted one or more message flow node properties, and want to delete them:

1. Switch to the Broker Application Development perspective.
2. Open the message flow for which you want to promote properties by double-clicking the message flow in the Navigator view. You can also open the message flow by right-clicking it in the Navigator view and clicking **Open**. The message flow contents are displayed in the editor view.

If this is the first message flow that you have opened, the message flow control window and the list of available built-in message flow nodes are also displayed, to the left of the editor view.

3. In the Editor view, right-click the symbol of the message flow node whose properties you want to promote.
4. Select **Promote Property**.

The Promote Property dialog is displayed.



5. Select the promoted property that you want to remove in the list of properties on the right of the dialog, and click **Remove**. The property is removed from the list on the right. It is restored to the list on the left, in its appropriate place in the tree of properties for the node from which you promoted it. You can promote this property again if you choose.
6. If you want to delete all the promoted properties within a single group, select the group in the list on the right and click **Remove**. The group and all the properties it contains are deleted from this list: the individual properties that you promoted are restored to the nodes from which you promoted them.
7. Click **Apply** to commit this change without closing the Property Promotion dialog. Click **OK** to complete your updates and close the dialog.

If you have included this message flow in a higher-level message flow, and have set a value for a promoted property that you have now deleted, the embedding flow is not automatically updated to reflect the deletion. However, when you deploy that embedding message flow in the broker domain, the deleted property is ignored.

Converging multiple properties

You can promote properties from several nodes in a message flow to define a single promoted property that provides a single value to be for that property in all those nodes. For example, if a message flow contains two Database nodes that each refer to the same physical database, you can define the physical database just once on the message flow by promoting the Data Source property of each Database node to the message flow, and setting the property at the message flow (promoted) level.

Before you start

To complete this task, you must have completed the following task:

- “Creating a message flow” on page 126

To converge multiple node properties to a single promoted property:

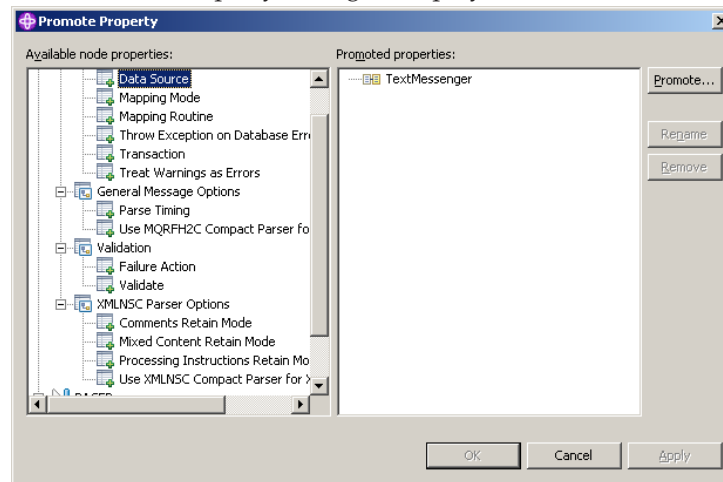
1. Switch to the Broker Application Development perspective.
2. Open the message flow for which you want to promote properties by double-clicking the message flow in the Navigator view. You can also open

the message flow by right-clicking it in the Navigator view and clicking **Open**. The message flow contents are displayed in the editor view.

If this is the first message flow that you have opened, the message flow control window and the list of available built-in message flow nodes are also displayed, to the left of the editor view.

3. In the editor view, right-click the symbol of the message flow node whose properties you want to promote.
4. Select **Promote Property**.

The Promote Property dialog is displayed.



5. Select the property that you want to converge. The list on the left initially shows the expanded list of all available properties for the selected node. If you have already promoted properties from this node, they do not appear on the left, but on the right.

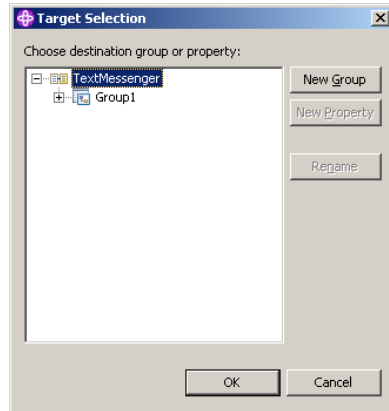
The list on the left also includes the other nodes in the open message flow. You can expand the properties listed under each node and work with all these properties at the same time. You do not have to close the dialog and select another node from the editor view to continue promoting properties.

You can select multiple properties to promote by selecting a property, holding down Ctrl, and selecting one or more other properties.

If you have you selected multiple properties to converge, all the properties you have selected must be available for promotion. If one or more of the selected properties is not available for promotion, the entire selection becomes unavailable for promotion, and the **Promote** button in the right-hand pane is grayed out.

6. Click the **Promote** button to promote the property or properties

Clicking the Promote button invokes the Target Selection dialog:



The Target Selection dialog displays only the valid targets for the promotion of the previously selected property or properties and allows you to create a new target for the promotion, such as to a new group or to a new property.

7. To converge properties from the same or different nodes in the message flow, expand the tree and click on a property that already exists. You can rename the properties by selecting them and clicking **Rename**, or by double-clicking on the group or property.
8. Click **OK** to confirm your selections.

Note: If you create a new group or property using the Target Selection dialog, the changes persist even if you select **Cancel** in the dialog. When the dialog closes, groups or properties that you have created using the Target Selection dialog will appear in the Promote properties dialog.

9. Expand the property trees for all the nodes for which you want to promote properties.
 - a. Drag the first instance of the property that you want to converge from the list on the left, and drop it on the appropriate group in the list on the right. If the group already contains one or more promoted properties, the new property is added at the end of the group. Rename the new property if you want to by double-clicking the property, or by selecting the property and clicking **Rename**.

If you want the promoted property to appear in a new group, you can drag and drop the property into an empty space below the existing groups, which forces a new group to be created. You can also place the promoted property in a new group by selecting the property you want to promote, and clicking **Promote**, which opens the Target Selection dialog. Click **New Group**, and enter the name of the new group. Click **OK** to confirm your changes.

If you drag the property onto an existing promoted property of a different type, a no-entry icon is displayed and you cannot drop the property. You must create this as a new promoted property, or drop it onto a compatible existing promoted property. Properties must be associated with the same property editor to be compatible. For example, if you are using built-in nodes, you can only converge like properties (string with string, boolean with boolean).

If you are using user-defined nodes, you must check the compatibility of the property editors for the properties that you want to converge. If you have written compiler classes for a node, you must also ensure that converged properties have the same compiler class.

10. Drag all remaining instances of the property from each of the nodes in the list on the left onto the existing promoted property. The new property is added under the existing promoted property, and is not created as a new promoted property.
11. Click **Apply** to commit this change without closing the Property Promotion dialog. Click **OK** to complete your updates and close the dialog.

Note: You can also converge properties from the Promote property dialog by dragging the selected property or properties from the left-hand pane of the Promote Property dialog to the right-hand pane:

1. Select the property you want to promote. You can select multiple properties to promote by selecting a property, holding down Ctrl, and selecting one or more other properties.
2. Drop the selected property or properties onto a property in the right-hand pane to converge related properties from the same or different nodes in the message flow.

For example, you might want to create a single promoted property that overrides the property on each node that defines a data source.

For more information on converging properties, see “Converging multiple properties” on page 371.

Collecting message flow accounting and statistics data

You can collect statistics on message flow behavior.

The following topics describe the tasks that you can complete to control collection of message flow accounting and statistics data:

- “Starting to collect message flow accounting and statistics data”
- “Stopping message flow accounting and statistics data collection” on page 377
- “Viewing message flow accounting and statistics data collection parameters” on page 378
- “Modifying message flow accounting and statistics data collection parameters” on page 379
- “Resetting message flow accounting and statistics archive data” on page 380

The topics listed here show examples of how to issue the accounting and statistics commands. The examples for z/OS are shown for SDSF; if you are using another interface, you must modify the example shown according to the requirements of that interface. For details of other z/OS options, see Issuing commands to the z/OS console.

Starting to collect message flow accounting and statistics data

Before you start:

Complete the following tasks:

- “Creating a message flow” on page 126
- “Deploying a broker archive file” on page 434

You can start collecting message flow accounting and statistics data for an active broker at any time.

Select the granularity of the data that you want to be collected by specifying the appropriate parameters on the `mqsichangeflowstats` command. You must request statistics collection on a broker basis. If you want to collect information for more than one broker, you must issue the corresponding number of commands.

To start collecting message flow accounting and statistics data:

1. Identify the broker for which you want to collect statistics .
2. Decide the resource for which you want to collect statistics. You can collect statistics for a specific execution group, or for all execution groups for the specified broker.
 - If you indicate a specific execution group, you can request that data is recorded for a specific message flow or all message flows in that group.
 - If you specify all execution groups, you must specify all message flows.
3. Decide if you want to collect thread related statistics.
4. Decide if you want to collect node related statistics. If you do, you can also collect information about terminals for the nodes.
5. Decide if you want to associate data collection with a particular accounting origin. This option is valid for snapshot and archive data, and for message flows and execution groups. However, when active, you must set its origin value in each message flow to which it refers. If you do not configure the participating message flows to set the appropriate origin identifier, the data collected for that message flow is collected with the origin set to `Anonymous`. See “Setting message flow accounting and statistics accounting origin” on page 376 for further details.
6. Decide the target destination:
 - User trace log. This is the default setting. The output data can be processed using `mqsireadlog` and `mqsiformatlog`.
 - XML format publication message. If you chose this as your target destination, register the following topic for the subscriber:

```
$/SYS/Broker/brokerName/StatisticsAccounting/recordType/executionGroupLabel/messageFlowLabel
```

Where, *brokerName*, *executionGroupLabel*, and *messageFlowLabel* are the broker, execution group and message flow on which you want to receive data. *recordType* is the type of data collection on which you want to receive publications (snapshot, archive, or # to receive both snapshot and archive).

- SMF (on z/OS only)
7. Decide the type of data collection that you want:
 - Snapshot
 - Archive

You can collect snapshot and archive data at the same time, but you have to configure them separately.

8. Issue the `mqsichangeflowstats` command with the appropriate parameters to reflect the decisions that you have made.

For example, to turn on snapshot data for all message flows in the default execution group for `BrokerA`, and include node data with the basic message flow statistics, enter:

```
mqsichangeflowstats BrokerA -s -e default -j -c active -n basic
```

Using SDSF on z/OS, enter:

```
/F BrokerA,cs s=yes,e=default,j=yes,c=active,n=basic
```

Refer to the `mqsichangeflowstats` command for further examples.

When the command completes successfully, data collection for the specified resources is started:

- If you have requested snapshot data, information is collected for approximately 20 seconds, and the results are written to the specified output.
- If you have requested archive data, information is collected for the interval defined for the broker (on the `mqsicreatebroker` or `mqsichangebroker` command, or by the external timer facility ENF). The results are written to the specified output, the interval is reset, and data collection starts again.

Next:

You can now perform the following tasks:

- “Setting message flow accounting and statistics accounting origin”
- “Stopping message flow accounting and statistics data collection” on page 377
- “Viewing message flow accounting and statistics data collection parameters” on page 378
- “Modifying message flow accounting and statistics data collection parameters” on page 379
- “Resetting message flow accounting and statistics archive data” on page 380

Setting message flow accounting and statistics accounting origin

Before you start:

Complete the following task:

- “Creating a message flow” on page 126

Accounting and statistics data is associated with an accounting origin.

When you request accounting origin support for collecting message flow accounting and statistics data on the `mqsichangeflowstats` command, you must also configure your message flows to provide the correct identification values that indicate what the data is associated with. You can set a different value for every message flow for which data collection is active, or the same value for a group of message flows (for example, those in a single execution group, or associated with a particular client, department, or application suite).

The accounting origin setting is not used until you deploy the message flow or flows to the brokers on which they are to run. You can activate data collection, or modify it to request accounting origin support, before or after you deploy the message flow. You do not have to stop collecting data when you deploy a message flow that changes accounting origin.

To configure a message flow to specify a particular accounting origin:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. Click **Selection** above the palette of nodes.
4. Right-click a Compute, Database, or Filter node in the editor view, and click **Open ESQL**. The associated ESQL file is opened in the editor view, and your cursor is positioned at the start of the correct module. You can include the required ESQL in any of these nodes, so decide which node in each message flow is the most appropriate for this action.

If you want to take advantage of the accounting origin support, you must include one of these nodes in each message flow for which you want a specific origin set. If you have not configured one of these three nodes in the message flow, you must add one at a suitable point (for example, immediately following the input node) and connect it to other nodes in the flow.

5. Update the ESQL in the node's module to set an accounting origin. The broker uses the origin identifier that is set in the Environment tree. You must set a value in the field with correlation name `Environment.Broker.Accounting.Origin`. This field is not created automatically in the Environment tree when the message is first received in the broker. It is created only when you set it in an ESQL module associated with a node in the message flow.

If you do not set a value in the message flow, the default value `Anonymous` is used for all output. If you set a value in more than one place in the message flow, the value that you set immediately before the message flow terminates is used in the output data.

The code that you need to add is of the form:

```
SET Environment.Broker.Accounting.Origin = "value";
```

You can set the identifier to a fixed value if you choose (as shown above), or you can determine its value based on a dynamic value that is known only at runtime. The value must be character data, and can be a maximum of 32 bytes. For example, you might set its value to the contents of a particular field in the message that is being processed (if you are coding ESQL for a Compute node, you must use correlation name `InputBody` in place of `Body` in the following example):

```
IF Body.DepartmentName <> NULL THEN  
  SET Environment.Broker.Accounting.Origin = Body.DepartmentName;  
END IF;
```

6. Save the ESQL module, and check that you have not introduced any errors.
7. Save the message flow, and check again for errors.

You are now ready to deploy the updated message flow. Accounting and statistics data records that are collected after the message flow has been deployed will include the origin identifier that you have set.

Stopping message flow accounting and statistics data collection

You can stop collecting data for message flow accounting and statistics at any time. You do not have to stop the message flow, execution group, or broker to make this change, nor do you have to redeploy the message flow.

Before you start:

Complete the following task:

- "Starting to collect message flow accounting and statistics data" on page 374

You can stop collecting data for message flow accounting and statistics at any time. You do not have to stop the message flow, execution group, or broker to make this change, nor do you have to redeploy the message flow.

You can modify the parameters that are currently in force for collecting message flow accounting and statistics data without stopping data collection. See “Modifying message flow accounting and statistics data collection parameters” on page 379 for further details.

To stop collecting data:

1. Check the resources for which you want to stop collecting data.

You do not have to stop all active data collection. If you choose, you can stop a subset of data collection. For example, if you started collecting statistics for all message flows in a particular execution group, you can stop doing so for a specific message flow in that execution group. Data collection for all other message flows in that execution group continues.

2. Issue the `mqsichangeflowstats` command with the appropriate parameters to stop collecting data for some or all resources.

For example, to switch off snapshot data for all message flows in all execution groups for BrokerA, enter:

```
mqsichangeflowstats BrokerA -s -g -j -c inactive
```

Using SDSF on z/OS, enter:

```
/F BrokerA,cs s=yes g=yes j=yes c=inactive
```

Refer to the `mqsichangeflowstats` command for further examples.

When the command completes successfully, data collection for the specified resources is stopped. Any outstanding data that has been collected is written to the output destination when you issue this command, to ensure the integrity of data collection.

Viewing message flow accounting and statistics data collection parameters

You can review and check the parameters that are currently in effect for message flow accounting and statistics data collection.

Before you start:

Complete the following task:

- “Starting to collect message flow accounting and statistics data” on page 374

To view message flow accounting and statistics data collection parameters:

Issue the `mqsireportflowstats` command with the appropriate parameters to view the parameters that are currently being used by the broker to control archive data collection or snapshot data collection.

You can view the parameters in force for a broker, an execution group, or an individual message flow.

For example, to view parameters for snapshot data for all message flows in all execution groups for BrokerA, enter:

```
mqsireportflowstats BrokerA -s -g -j
```

Using SDSF on z/OS, enter:

```
/F BrokerA,rs s=yes,g=yes,j=yes
```

Refer to the `mqsireportflowstats` command for further examples.

The command displays the current status, for example:

```
BIP8187I: Statistics Snapshot settings for flow MyFlow1 in execution
group default - On?: inactive,
ThreadDataLevel: basic, NodeDataLevel: basic,
OutputFormat: usertrace, AccountingOrigin: basic
```

Next:

You can now modify the data collection parameters.

Modifying message flow accounting and statistics data collection parameters

You can modify the parameters that you have set for message flow accounting and statistics data collection. For example, you can start collecting data for a new message flow that you have deployed to an execution group for which you are already collecting data.

You can modify parameters while data collection is active; you do not have to stop data collection and restart it.

Before you start:

Complete the following task:

- “Starting to collect message flow accounting and statistics data” on page 374

To modify message flow accounting and statistics parameters:

1. Decide which data collection parameters you want to change. You can modify the parameters that are in force for a broker, an execution group, or an individual message flow.
2. Issue the `mqsichangeflowstats` command with the appropriate parameters to modify the parameters that are currently being used by the broker to control archive data collection or snapshot data collection.

For example, to modify parameters to extend snapshot data collection to a new message flow MFlow2 in execution group EG2 for BrokerA, enter:

```
mqsichangeflowstats BrokerA -s -e EG2 -f MFlow2 -c active
```

Using SDSF on z/OS, enter:

```
/F BrokerA,cs s=yes,e=EG2,f=MFlow2,c=active
```

If you want to specify an accounting origin for archive data for a particular message flow in an execution group, enter:

```
mqsichangeflowstats BrokerA -a -e EG4 -f MFlowX -b basic
```

Using SDSF on z/OS, enter:

```
/F BrokerA,cs a=yes,e=EG4,f=MFlowX,b=basic
```

Refer to the `mqsichangeflowstats` command for further examples.

When the command completes successfully, the new parameters that you have specified for data collection are in force. These parameters remain in force until you stop data collection or make further modifications.

Resetting message flow accounting and statistics archive data

You can reset message flow accounting and statistics archive data to purge any accounting and statistics data not yet reported for that collecting interval. This removes unwanted data. You can request this at any time; you do not have to stop data collection and restart it to perform reset. You cannot reset snapshot data.

Before you start:

Complete the following task:

- “Starting to collect message flow accounting and statistics data” on page 374

To reset message flow accounting and statistics archive data:

1. Identify the broker, and optionally the execution group, for which you want to reset archive data. You cannot reset archive data on a message flow basis.
2. Issue the `mqsichangeflowstats` command with the appropriate parameters to reset archive data.

For example, to reset archive data for BrokerA, enter:

```
mqsichangeflowstats BrokerA -a -g -j -r
```

Using SDSF on z/OS, enter:

```
/F BrokerA,cs a=yes,g=yes,j=yes,r=yes
```

When this command completes, all accounting and statistics data accumulated so far for this interval are purged and will not be included in the reports. Data collection is restarted from this point. All archive data for all flows (indicated by `-j` or `j=yes`) in all execution groups (indicated by `-g` or `g=yes`) is reset.

This command has a minimal effect on snapshot data because the accumulation interval is much shorter than for archive data. It does not effect the settings for archive or snapshot data collection that are currently in force. When the command has completed, data collection resumes according to the current settings.

You can change any other options that are currently in effect when you reset archive data, for example accounting origin settings or output type.

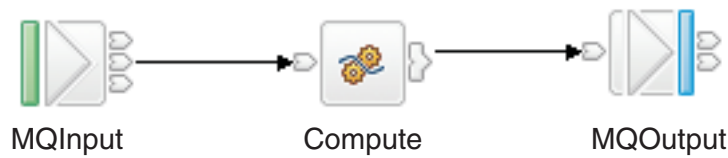
Developing user exits

User exits enable user-provided custom software to track data passing through message flows in WebSphere Message Brokers.

The user-provided functions can be invoked at specific points during the life-cycle of a message as it passes through the message flow, and can invoke utility functions to query information about the point in the flow, and the contents of the message assembly.

The user exits can be invoked when one or more of the following events occurs:

- End of unit-of-work (UOW) or transaction (COMMIT/ROLLBACK)
- A message passing between two nodes
- A message dequeued from the input source.



In the basic message flow shown above, you can track messages at two levels:

- Transaction level
- Node level

At the Transaction level, you can track the following events:

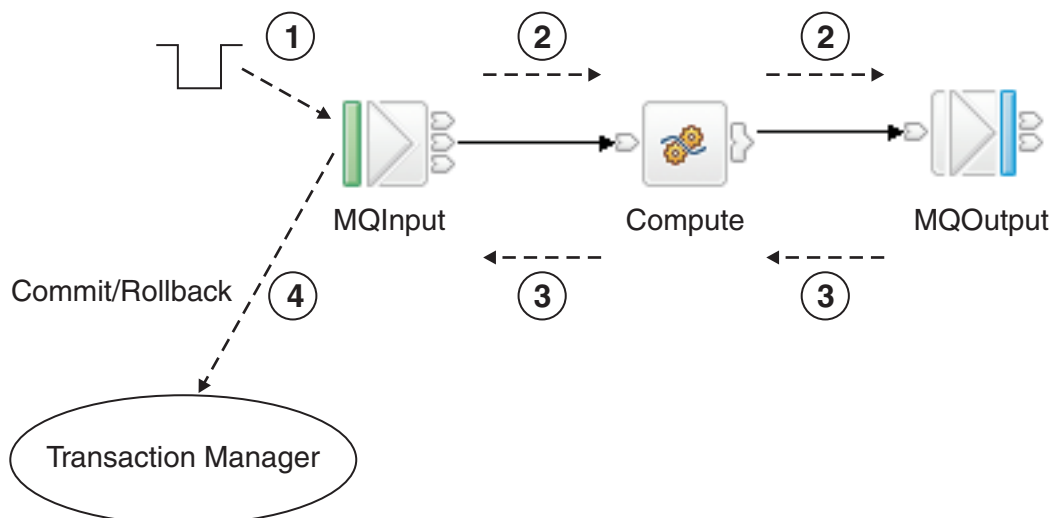
- Messages being read into the flow
- Completion of the transaction.

At the Node level, you can track the following events:

- A message passing from one node to another
- Completion of processing for one node.

This means that there are four different types of events, which occur in the following sequence:

1. Message dequeued from the input source (read into the flow)
2. Message propagated to the node for processing
3. Node processing completed
4. End of the transaction.



In the diagram above, the MQInput node is used as an example, but the function applies to all input nodes, including user-defined nodes. However, there is a slight difference between built-in nodes and user-defined nodes, in the way in which user exits are invoked. For built-in input nodes, the user exit is invoked as soon as

possible after the data has been read from the external source. For user-defined input nodes, the user exit is called just before the node propagates the message.

In the example shown above, event 4 is fired at the end of the transaction. The user exit is invoked after the transaction has completed, so the user exit processing is not part of that transaction. The user exit is invoked even if no transactional processing was completed by the flow. Where the message flow property Commit Count is greater than 1, there is a many-to-one ratio between event 1 and event 4. This is also true for some scenarios specific to the particular input node; for example, when an MQInputNode is configured with the Commit by Message Group property selected.

You can write a user exit to track any number of these events. For each of these events, the following data is available to the user exit. All access is read-only, unless stated otherwise:

- Dequeue message:
 - Bitstream
 - Input node
 - Environment tree (read/write)
- Message propagated to node:
 - Message tree
 - LocalEnvironment tree (read/write)
 - Exception list
 - Environment tree (read/write)
 - Source node
 - Target node
- Node processing completed:
 - Message tree
 - LocalEnvironment tree (read/write)
 - Exception list
 - Environment tree (read/write)
 - Node
 - Upstream node
 - Exception (if any)
- End of transaction:
 - Input node
 - Exception if any
 - Environment tree (read/write)

Multiple user exits can be registered, and, if they are, they are invoked in a defined order (see `mqsischangeuserexit` command). Any changes made to the message assembly by a user exit are visible to subsequent user exits.

When the user exit is invoked, the following information can be queried:

- Message flow information:
 - Message flow name
 - Broker name
 - Broker's queue manager name
 - Execution group name

- Message flow’s commit count property
- Message flow’s commit interval property
- Message flow’s coordinated transaction property.
- Node information:
 - Node name
 - Node type
 - Terminal name
 - Node properties
- Navigate and read the message assembly (Message,LocalEnvironment,ExceptionList)
- Navigate and read/write the Environment tree.

The user exits can be registered on a dynamic basis, without needing to redeploy the configuration.

Developing a user exit

To develop a user exit, follow these steps:

1. Declare the user exit

You declare a user exit by using the `bipInitializeUserExits` function to specify the following things:

- a. Name (used to register and control the active state of the exit)
- b. User context storage
- c. A function to be invoked (for one or more Event Types)

2. Implement the user exit behavior

When the user exit is declared, a set of functions is registered, and these functions are invoked when specific events occur. The behavior of the user exit is provided by implementing these functions. The following table lists the events and their associated functions:

Event	Function
Message dequeued from input source	<code>cciInputMessageCallback</code>
Message propagated to node for processing	<code>cciPropagatedMessageCallback</code>
Node completed processing	<code>cciNodeCompletionCallback</code>
Transaction ended	<code>cciTransactionEventCallback</code>

3. Implement the cleanup function

The user exit library must implement the `bipTerminateUserExits` function. This function is invoked as the `ExecutionGroup`’s process is ending, which allows you to clear up any resources allocated during the `bipInitializeUserExits` function.

Deploying a user exit

When you have written and compiled the user exit, you need to give the library the extension “.lel”, export the functions `bipInitializeUserExits` and `bipTerminateUserExits`, and install the library on the broker’s system. Additionally, the state of the user exit can be set to active or inactive on a per message flow basis.

To deploy the user exit:

1. Install the user exit code on a broker

The library containing the user exit code must be installed on a file system that can be accessed by the broker. For example, the file must have read and execute authority for the user ID under which the broker runs. The broker looks in the following places for libraries containing user exits:

- The broker property UserExitPath (UserExitPath64 for a 64-bit execution group). This is a list of directories separated by colons (semi-colons on Windows). It can be set using the `-x` flag on `mqsicreatebroker` or `mqsicchangebroker`. Set this to load a user exit into specific brokers.
- The environment variable MQSI_USER_EXIT_PATH (MQSI_USER_EXIT_PATH64 for a 64-bit execution group). This is a list of directories separated by colons (semi-colons on Windows). This is typically set to load the user exit into every broker for a specific environment.

If both are set, the environment variable takes precedence. All directories in the environment variable are searched, in the order in which they appear in the variable. Then, all directories in the broker property are searched, in the order in which they appear in the property.

2. Load the user exit library into the broker's processes

When the user exit library has been installed on the broker, it must be loaded. This can be done in either of the following ways:

- Stop and restart the broker
- Issue the `mqsireload` command. This causes the execution group processes to be restarted.

3. Activate the user exit

User exits can be active or inactive, and are inactive by default. The default state for a set of user exits can be changed to active on a per-broker basis.

To set the default user exit state for a broker:

- a. Stop the broker
- b. Set the `activeUserExits` property of the broker using the `mqsicchangebroker` command.
- c. Start the broker and check the system log to ensure that all execution groups start without error. If any invalid user exit names are specified (that is, the user exit is not provided by any library loaded by the execution group) a BIP2314 message is written to the system log and all flows in the execution groups fail to start.

You can also override the default user exit state for a broker. User exits can be activated or deactivated on a per-execution group or message flow basis, using the `mqsichangeuserexit` command, with the order of precedence being message flow then execution group. When multiple exits are active for a given flow, they are invoked in a defined order (as described in `mqsichangeuserexit`).

Configuring aggregation flows

Aggregation message flows let you generate and fan-out a number of related requests, fan-in the corresponding replies, and compile those replies into a single aggregated reply message, using the `AggregateControl`, `AggregateRequest`, and `AggregateReply` nodes. For an overview of using aggregation in message flows, see "Message flow aggregation" on page 90.

To configure aggregation flows see the following topics:

- “Creating the aggregation fan-out flow”
- “Creating the aggregation fan-in flow” on page 389
- “Associating fan-out and fan-in aggregation flows” on page 393
- “Setting timeouts for aggregation” on page 394
- “Using multiple AggregateControl nodes” on page 395
- “Correlating input request and output response aggregation messages” on page 396
- “Using control messages in aggregation flows” on page 397
- “Handling exceptions in aggregation flows” on page 400

The Airline sample demonstrates the use of aggregation message flows.

Creating the aggregation fan-out flow

The aggregation fan-out flow receives the initial input message and restructures it to present a number of requests to a number of target applications.

Before you start:

To complete this task, you must have completed the following task:

- “Creating a message flow project” on page 123

You are also advised to read the overview about “Message flow aggregation” on page 90, before completing this task.

It is possible to include the fan-out and fan-in flow within the same message flow. However it is advisable to create two separate flows. For more information about the benefits of configuring separate message flows, see “Associating fan-out and fan-in aggregation flows” on page 393.

To review an example of a fan-out flow, see the Airline Reservations sample that is supplied with WebSphere Message Broker.

To create the fan-out flow:

1. Switch to the Broker Application Development perspective.
2. Create a new message flow to provide the fan-out processing.
3. Add the following nodes in the editor view and configure and connect them as described:

Input node

The input node receives an input message from which multiple request messages are generated. This can be any one of the built-in nodes, or a user-defined input node.

- a. Right-click the input node and click **Properties**.
- b. Specify the source of input messages for this node. For example, specify the name of a WebSphere MQ queue in the Basic property *Queue Name* from which the MQInput node retrieves messages.
- c. Optional: specify values for any other properties that you want to configure for this node. It is recommended that you set the Advanced property *Transaction mode*, to the default, Yes, to ensure that aggregate request messages are put under syncpoint. This will avoid a situation where the AggregateReply node receives response messages before it has received the control message informing it of the aggregation instance. By putting the fan-out flow under

transactional control you ensure that the fan-out flow completes before any response messages get to the `AggregateReply`.

- d. Connect the input node's out terminal to the in terminal of an `AggregateControl` node.

This represents the simplest configuration; if appropriate, you can include other nodes between the input node and the `AggregateControl` node. For example, you might want to store the request for audit purposes (in a `Warehouse` node), or add a unique identifier to the message (in a `Compute` node).

- e. Optional: If your fan-out and fan-in flows are combined within one message flow, it is advisable to modify the *Order Mode* property on the Advanced tab. Select the **By Queue Order** option and ensure that the **Logical Order** option is also selected. This forces the input node to be single threaded so as to maintain the logical order of the messages that arrive on the queue. This results in any additional instance threads that you make available, being shared amongst only the fan-in input nodes to improve the performance of aggregation. If your fan-in and fan-out flows are in separate message flows this step is not necessary as you can make additional threads available specifically to the fan-in flow.

AggregateControl node

The `AggregateControl` node updates the `LocalEnvironment` associated with the input message with information required by the `AggregateRequest` node.

- a. Right-click the `AggregateControl` node and click **Properties**.
- b. Set the *Aggregate Name* property of the `AggregateControl` node to identify this particular aggregation. It is used later to associate this `AggregateControl` node with a specific `AggregateReply` node. The *Aggregate Name* that you specify must be contextually unique within a broker.
- c. Optional: set the *Timeout* property to specify how long the broker waits for replies to arrive before taking some action (described in "Setting timeouts for aggregation" on page 394).

If a timeout is not set on the `AggregateControlNode` then aggregate requests stored internally will not be removed unless all aggregate reply messages return. This might lead to a gradual build up of messages on the internal queues. To avoid this situation, set the timeout to a value other than zero (zero means never timeout) so that when the timeout is reached the requests are removed and the queues will not fill up with redundant requests. Even if timeouts are not required or expected, it is good practice to set the timeout value to a large value (for example: 864000 seconds which is 24 hours) so that the queues occasionally get cleared of old aggregations.

- d. Connect the out terminal of the `AggregateControl` node to the in terminal of one or more `Compute` nodes that provide the analysis and breakdown of the request in the input message that is propagated on this terminal.

Note: The control terminal of the `AggregateControl` node has been deprecated at Version 6.0 and by default any connections from this terminal to the `AggregateReply` node (either direct or indirect), will be ignored. This is to maximize the efficiency of

aggregation flows and does not damage the reliability of aggregations. This is the optimum configuration.

However, if you do want a control message to be sent from the AggregateControl node to the AggregateReply node, you must connect the control terminal to the corresponding AggregateReply node on the fan-in flow (either directly or indirectly, as described in “Associating fan-out and fan-in aggregation flows” on page 393). If you connect it indirectly to the AggregateReply node, for example through an MQOutput node, you must include a Compute node to add the appropriate headers to the message to ensure that it can be safely transmitted.

In addition, for the Control terminal and connections from it to be recognized, you must enable the environment variable MQSI_AGGR_COMPAT_MODE. However, choosing this option has implications regarding the performance and behavior of message aggregations. For a full description of these implications and the environment variable, see “Using control messages in aggregation flows” on page 397.

Compute node

The Compute node extracts information from the input message and constructs a new output message.

If the target applications that handle the subtask requests can extract the information that they require from the single input message, you do not need to include a Compute node to split the message. You can pass the whole input message to all target applications.

If your target applications expect to receive an individual request, not the whole input message, you must include a Compute node to generate each individual subtask output message from the input message. Configure each Compute node in the following way, copying the appropriate subset of the input message to each output message.

- a. Right-click the Compute node and click **Properties**.
- b. Select a value for the Basic property *Compute Mode*. This property specifies which sections of the message tree are modified by the node.

The AggregateControl node inserts elements into the LocalEnvironment tree in the input message that the AggregateRequest node reads when the message reaches it. Ensure that the LocalEnvironment is copied from the input message to the output message in the Compute node. This happens automatically unless you specify a value that includes LocalEnvironment (one of All, LocalEnvironment, LocalEnvironment and Message, or Exception and LocalEnvironment).

If you specify one of these values, the broker assumes that you are customizing the Compute node with ESQL that writes to LocalEnvironment, and that you will copy over any elements within that tree that are required in the output message.

If you want to modify LocalEnvironment, add the following statement to copy the required aggregate information from input message to output message:

```
SET OutputLocalEnvironment.ComIbmAggregateControlNode =  
    InputLocalEnvironment.ComIbmAggregateControlNode;
```

- c. Optional: specify values for any other properties that you want to configure for this node.
- d. Connect the out terminal of each Compute node to the in terminal of the output node that represents the destination of the output request message that you have created from the input message in this node.

Output node

Include an output node for each output message that you generate in your fan-out flow. Configure each node as described below, with the appropriate modifications for each destination.

This must be an output node that supports the request/reply model, such as an MQOutput node, or a mixture of these nodes (depending on the requirements of the target applications).

- a. Right-click the output node and click **Properties**.
- b. Specify the destination for the output messages for this node. For example, specify the name of a WebSphere MQ queue in the Basic property *Queue Name* to which the MQOutput node sends messages. The target application must process its request, and send the response to the reply destination indicated in its input message (for example the WebSphere MQ ReplyToQueue).
- c. Click Request in the left view and set values for these properties to specify that replies are sent to the fan-in flow's input queue.
- d. Optional: specify values for any other properties that you want to configure for this node.
- e. Connect the out terminal of the output node to the in terminal of an AggregateRequest node. When the message is propagated through the output node's out terminal, the built-in output node updates the WrittenDestination folder within the associated LocalEnvironment with additional information required by the AggregateRequest node. The information written by the built-in nodes is queue name, queue manager name, message ID and correlation ID (from the MQMD), and message reply identifier (set to the same value as message ID).

AggregateRequest node

Include an AggregateRequest node for each output message that you generate in your fan-out flow.

- a. Right-click the output node and click **Properties**.
- b. Set the Basic property *Folder Name* to a value that identifies the type of request that has been sent out. This value is used by the AggregateReply node to match up with the reply message when it is received in the fan-in flow. The folder name that you specify for each request that the fan-out flow generates must be unique.

The AggregateRequest node writes a record in the broker database for each message that it processes. This enables the AggregateReply node to identify which request each response is associated with. If your output nodes are non-transactional, the response message might arrive at the fan-in flow before this database update is committed. Refer to "Setting timeouts for aggregation" on page 394 for details on how you can use timeouts to avoid this.

CAUTION:

Although the use of timeouts can help to avoid this situation described above, it is preferable to ensure that it is not possible for response messages to get to the fan-in flow before the corresponding AggregateRequest nodes have committed their database updates, by making your fan-out flow transactional.

4. Press Ctrl-S to save the message flow and validate its configuration.

To collect the aggregation responses initiated by your fan-out flow, create you fan-in flow, see “Creating the aggregation fan-in flow.”

Creating the aggregation fan-in flow

The aggregation fan-in flow receives the responses to the request messages sent out by the fan-out flow and constructs a combined response message containing all the responses received.

Before you start:

To complete this task, you must have completed the following task:

- “Creating a message flow project” on page 123

You are also advised to read the overview about “Message flow aggregation” on page 90, before completing this task.

Depending on whether the fan-out flow is transactional and, if not, the timeout values that you have specified, the combined response message might be generated before all the replies have been received by the fan-in flow. See, “Creating the aggregation fan-out flow” on page 385 for more information on this topic.

It is possible to include the fan-out and fan-in flow within the same message flow. However it is advisable to create two separate flows. Do not deploy multiple copies of the same fan-in flow either in the same or different execution groups. For more information about the benefits of configuring separate message flows, see “Associating fan-out and fan-in aggregation flows” on page 393.

To review an example of a fan-in flow, see the Airline Reservations sample that is supplied with WebSphere Message Broker.

To create the fan-in flow:

1. Switch to the Broker Application Development perspective.
2. Create a message flow to provide the fan-in processing.
3. Add the following nodes in the editor view and configure and connect them as described:

Input node

The input node receives the responses to the multiple request messages generated from the fan-out flow.

This must be an input node that supports the request/reply model, such as an MQInput node, or a mixture of these nodes (depending on the requirements of the applications that send these responses). The response received by each input node must be sent across the same protocol as the request to which it corresponds (for example, if you include an MQOutput node in the fan-out flow, the response to that request must be received by an MQInput node in this flow).

- a. Right-click the input node and click **Properties**.
- b. Specify the source of input messages for this node. For example, specify the name of a WebSphere MQ queue in the Basic property *Queue Name* from which the MQInput node retrieves messages.
- c. Optional: specify values for any other properties that you want to configure for this node.
- d. Connect the input node's out terminal to the in terminal of an AggregateReply node.

This represents the simplest configuration; if appropriate, you can include other nodes between the input node and the AggregateReply node. For example, you might want to store the replies for audit purposes (in a Warehouse node).

Note: It is advisable to have one input node which receives all of the aggregation response messages at the beginnings of the fan-in flow as described above. Using one instead of multiple nodes ensures that no specific reply input node thread always gets to complete the aggregation and execution of the message flow, whilst the others send their response messages to the Aggregate Reply, which subsequently become eligible to timeout. Using one single input node creates a more sequential processing of the replies for each aggregation which can be successfully scaled by using additional instances to provide more processing power.

AggregateReply node

The AggregateReply node receives the inbound responses from the input node through its in terminal. Each reply message received by the AggregateReply node is stored persistently in the broker database.

When all the replies for a particular group of aggregation requests have been collected, the AggregateReply node creates an aggregated reply message and propagates this through the out terminal.

- a. Right-click the AggregateReply node and click **Properties**.
- b. Set the *Aggregate Name* property of the AggregateReply to identify this aggregation. Set this value to be the same value that you set for the *Aggregate Name* property in the corresponding AggregateControl node in the fan-out flow.
- c. Optional: set a value for the *Unknown Message Timeout* if you want to retain an unrecognized message before propagating it to the unknown terminal. If you are using separate fan-out and fan-in flows, you might want to set this value to a non-zero number in case there are delays in the arrival of the control message.
- d. Optional: connect the unknown terminal to another node or sequence of nodes if you want to explicitly handle unrecognized messages. If you do not connect this terminal to another node in the message flow, messages propagated through this terminal are discarded.
- e. Optional: if you have specified a timeout value for this aggregation in the AggregateControl node, connect the timeout terminal to another node or sequence of nodes if you want to explicitly handle timeouts that expire before all replies are received. Partially complete aggregated replies are sent to the timeout terminal if the timer expires. If you do not connect this terminal to another node in the message flow, messages propagated through this terminal are discarded.

- f. Optional: specify values for any other properties that you want to configure for this node.
- g. Connect the out terminal of the AggregateReply node to the in terminal of a Compute node.

Note: The Control terminal of the AggregateReply node has been deprecated at Version 6.0 and by default any connections to this terminal (either direct or indirect), will be ignored. This is to maximize the efficiency of aggregation flows and does not damage the reliability of aggregations. This is the optimum configuration.

However, if you do want the AggregateReply node to receive the control message on its control terminal that was sent by the corresponding AggregateControl node on the fan-out flow, you must make the necessary connections as described in “Creating the aggregation fan-out flow” on page 385. It is recommended that the path from the AggregateReply node to the output node is as direct as possible. Keeping the fan-in flow short, will benefit the performance of aggregations. Do not modify the content of this control message.

In addition, for the Control terminal and connections to it to be recognized, you must enable the environment variable MQSI_AGGR_COMPAT_MODE. However, choosing this option has implications regarding the performance and behavior of message aggregations. For a full description of these implications and the environment variable, see “Using control messages in aggregation flows” on page 397.

Compute node

The Compute node receives the message that contains the combined responses. It is unlikely that this combined message is in a format that is valid for output, because the aggregated reply message has an unusual structure and cannot be turned into the bitstream required by some nodes, for example the MQOutput node. The out and timeout terminals always output an aggregated reply message, so always require further processing before an MQOutput node. Therefore you must include a compute node and configure this node to create a valid output message.

- a. Right-click the Compute node and click **Properties**.
- b. Specify the name of the ESQL module that customizes the function of this node in the Basic property *ESQL Module*.
- c. Right-click the node and click **Open ESQL** to open the ESQL file that contains the module for this node. The module is highlighted in the ESQL editor view.
- d. Code the ESQL to create a single output message from the aggregated replies in the input message.

The structure of the aggregated reply message that is propagated through the out terminal, and information on how you can access its contents, are provided in “Accessing the combined message contents” on page 392.

- e. Optional: specify values for any other properties that you want to configure for this node.

- f. Connect the out terminal of the Compute node to the in terminal of the output node that represents the destination of the single response message.

Output node

Include an output node for your fan-in flow. This can be any of the built-in nodes, or a user-defined output node.

- a. Right-click the output node and click **Properties**.
 - b. Specify the destination for the output message for this node. For example, specify the name of a WebSphere MQ queue in the Basic property *Queue Name* to which the MQOutput node sends messages.
 - c. Optional: specify values for any other properties that you want to configure for this node.
4. Press Ctrl-S to save the message flow and validate its configuration.

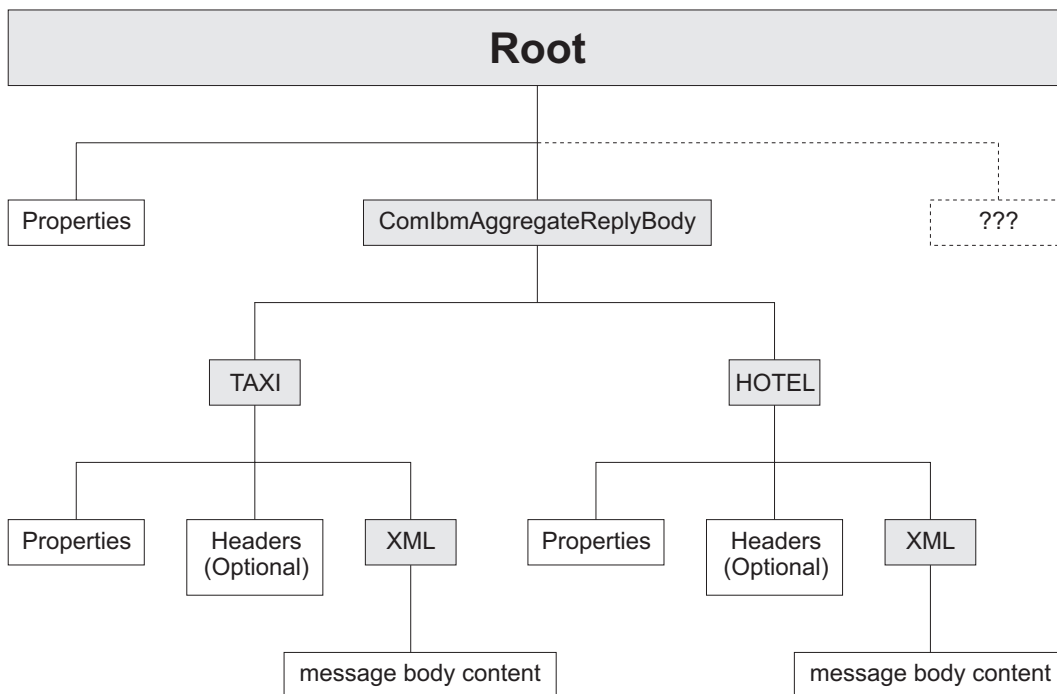
Accessing the combined message contents

The AggregateReply node creates a folder in the combined message tree below Root, called ComIbmAggregateReplyBody. Below this, it creates a number of folders using the folder names that you set in the AggregateRequest nodes. The associated reply messages are put beneath them.

For example, the request messages might have folder names:

- TAXI
- HOTEL

The resulting aggregated reply message created by the AggregateReply node might have a structure similar to that shown below:



You can use a Compute node to access the reply from the taxi company using the following correlation name:

```
InputRoot.ComIbmAggregateReplyBody.TAXI.xyz
```

The folder name does not have to be unique. If you have multiple requests with the folder name TAXI, you can access the separate replies using the array subscript notation, for example:

```
InputRoot.ComIbmAggregateReplyBody.TAXI[1].xyz  
InputRoot.ComIbmAggregateReplyBody.TAXI[2].xyz
```

Associating fan-out and fan-in aggregation flows

Associate the fan-out message flow processing with its corresponding fan-in message flow processing by setting the *Aggregate Name* property of the `AggregateControl` and `AggregateReply` nodes in your aggregation flow to the same value. If you did not configure this property whilst creating your fan-in and fan-out flows you must complete this task.

Before you start:

To complete this task, you must have completed the following tasks:

- “Creating the aggregation fan-out flow” on page 385
- “Creating the aggregation fan-in flow” on page 389

The *Aggregate Name* must be contextually unique within a broker. In general, this means that there should only be one `AggregateControl` node and one `AggregateReply` node with a particular *Aggregate Name*, although it is possible to have more than one `AggregateControl` node with the same *Aggregate Name*, as described in “Using multiple `AggregateControl` nodes” on page 395. It is not advisable to deploy a fan-in flow to multiple execution groups on the same broker.

You can either create the fan-out and fan-in flows in the same message flow, or in two different message flows. In either case, the two parts of the aggregation are associated by setting the *Aggregate Name* property.

The way in which you configure your aggregation flow depends on a number of factors:

- The design of your message flow
- The hardware on which the broker is running
- The timeout values that you choose (see “Setting timeouts for aggregation” on page 394)
- How you expect to maintain the message flows

A single flow is easier to implement for a simple case, but there are some limitations to this approach, and, in most cases, you will find that the flexibility offered by two message flows is preferable. The `sampleAirlineReservations` sample shows the use of two flows for aggregation.

The advantages of creating separate fan-out and fan-in flows are:

- The two flows can be modified independently of each other.
- The two flows can be stopped and started independently of each other.

- The two flows can be deployed to separate execution groups to take advantage of multiprocessor systems, or to provide data segregation for security or integrity purposes.
- The two flows can be assigned different numbers of additional threads as appropriate to maintain an appropriate processing ratio.

To associate the fan-out flow with the fan-in flow:

1. Open the message flow containing your fan-out flow.
2. Right-click the AggregateControl node and click **Properties**.
3. Set the *Aggregate Name* property of the AggregateReply node to identify this aggregation. The *Aggregate Name* that you specify must be contextually unique within a broker.
4. If you have separate fan-out and fan-in flows:
 - a. Press Ctrl-S to save the fan-out message flow and validate its configuration.
 - b. Open the message flow containing your fan-in flow.
5. Right-click the AggregateReply node and click **Properties**.
6. Set the *Aggregate Name* property of the AggregateReply node to the same value that you set for the *Aggregate Name* property in the corresponding AggregateControl node in the fan-out flow.
7. Press Ctrl-S to save the message flow and validate its configuration.

In Version 5.0 of the product, fan-out and fan-in flows were also associated by sending control messages from the AggregateControl node to the AggregateReply node. This is no longer necessary. For optimum performance it preferable not to connect the AggregateControl and AggregateReply node. However, if you do want to use control messages in your aggregations and connect these node see, “Using control messages in aggregation flows” on page 397.

Note: Connections between the AggregateControl and AggregateReply node created in Version 5.0 will be ignored once you have migrated them to version 6.0 unless you set the broker environment to specify otherwise. For more information see, “Using control messages in aggregation flows” on page 397.

Setting timeouts for aggregation

You can use two properties of the aggregation nodes to set timeout values for aggregated message processing.

Before you start:

To complete this task, you must have completed the following tasks:

- “Creating the aggregation fan-out flow” on page 385
- “Creating the aggregation fan-in flow” on page 389

There are two situations that might require the use of timeouts:

1. In certain situations you might need to receive an aggregated reply message within a certain time. Some reply messages might be slow to return, or might never arrive. For these situations:
 - a. Switch to the Broker Application Development perspective.
 - b. Open the fan-out message flow.

- c. Set the *Timeout* property of the `AggregateControl` node to specify how long (in seconds) the broker must wait for replies. By default, this property is set to 0, which means that there is no timeout and the broker waits indefinitely.

If the timeout interval passes without all the replies arriving, the replies that have arrived are turned into an aggregated reply message by the corresponding `AggregateReply` node, and propagated to its timeout terminal. If you choose, you can process this partial response message in the same way as a complete aggregated reply message. If you prefer, you can provide special processing for incomplete aggregated replies.

2. When a message arrives at the in terminal of an `AggregateReply` node, it is examined to see if it is an expected reply message. If it is not recognized, it is propagated to the unknown terminal. You might want the broker to wait for a given period of time before doing this, because:
 - The reply message might arrive before the work performed by the `AggregateRequest` node has been transactionally committed. This situation can be avoided by configuring the *Transaction mode* property of the `Input` node as described in “Creating the aggregation fan-out flow” on page 385.
 - The reply message might arrive before the control message. This situation can be avoided by leaving the control terminal of the `AggregateControl` node unconnected. For further information about the implications of connecting the control terminal, see “Using control messages in aggregation flows” on page 397.

These situations are most likely to happen if you send the request messages out of syncpoint, and might result in valid replies being sent to the unknown terminal. To reduce the chance of this event:

- a. Switch to the Broker Application Development perspective.
- b. Open the fan-in message flow.
- c. Set the *Unknown Message Timeout* property on the `AggregateReply` node. When you set this property, a message that cannot be recognized immediately as a valid reply is held persistently within the broker for the number of seconds that you specify for this property .

If the unknown timeout interval expires, and the message is recognized, it is processed. The node also checks to see if this previously unknown message is the last reply needed to make an aggregation complete. If it is, the aggregated reply message is constructed and propagated.

If the unknown timeout interval expires and the message is still not recognized, the message is propagated to the unknown terminal.

Using multiple `AggregateControl` nodes

You might find it useful to design a fan-out flow with multiple `AggregateControl` nodes, all with the same value set for the property *Aggregate Name*, but with different values for the *Timeout* property. This is the only situation in which you can reuse an `Aggregate Name`.

Before you start:

To complete this task, you must have completed the following task:

- “Creating a message flow project” on page 123

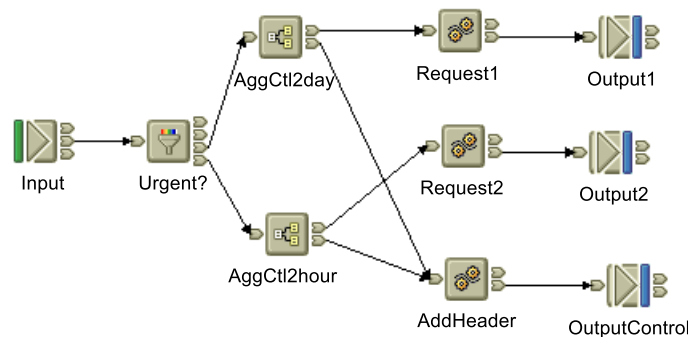
For example, if you have created an aggregation flow that books a business trip, you might have some requests that need a reply within two days, but other, more urgent requests, that need a reply within two hours.

To configure an aggregation flow that uses multiple AggregateControl nodes:

1. Switch to the Broker Application Development perspective.
2. Create or open the fan-out message flow.
3. Configure the required number of AggregateControl nodes. Set the Basic property *Aggregate Name* of each node to the same value. For example, include two nodes and enter the name JOURNEY as the *Aggregate Name* for both.
4. Set the value for the *Timeout* property in each node to a different value. For example, set the *Timeout* in one node to two hours; set the *Timeout* in the second node to two days.
5. Configure a Filter node to receive incoming requests, check their content, and route them to the correct AggregateControl node.
6. Connect the nodes together to achieve the required result. For example, if you have configured the Filter node to test for requests with a priority field set to urgent, connect the true terminal to the AggregateControl node with the short timeout. Connect the false terminal to the AggregateControl node with the longer timeout. Connect the out terminals of the AggregateControl nodes to the following nodes in the fan-out flow.

You must connect the two AggregateControl nodes in parallel, not in sequence. This means that you must connect both to the Filter node (one to the true terminal, one to the false), and both to the downstream nodes that handle the requests for the fan-out. Each input message must pass through only one of the AggregateControl nodes. If you connect the nodes such that a single message is processed by more than one AggregateControl node, duplicate records are created in the database by the AggregateRequest node and subsequent processing results are unpredictable.

The following diagram shows an example fan-out message flow that uses this technique.



Correlating input request and output response aggregation messages

If you want to correlate initial request messages with their combined response messages, you can do so using the ReplyIdentifier in the Properties folder of the response message.

Before you start:

To complete this task, you must have completed the following tasks:

- “Creating the aggregation fan-out flow” on page 385
- “Creating the aggregation fan-in flow” on page 389

In some cases you might want to correlate aggregation request messages with the combined response message produced by your fan-in flow, there are two ways of doing this:

- Store some correlation information in one of the requests sent out as part of the aggregation.
- Send the original request message directly back to the AggregateReply node as one of the aggregation requests. To do this, the CorrelId must be set to the MsgId, and the MQOutput node must have its MessageContext set to ‘Pass all’.

Using control messages in aggregation flows

Before you start:

To complete this task, you must have completed the following tasks:

- “Creating the aggregation fan-out flow” on page 385
- “Creating the aggregation fan-in flow” on page 389

The following topic describes the actions required to send control messages. In WebSphere Message Broker the default behavior is that connections between AggregateControl and AggregateReply nodes for sending control messages will be ignored. This configuration optimizes performance and removes the possibility that response messages will be received by the AggregateReply node before the control message. Control messages are not necessary to make aggregations work correctly. However, if it is still necessary for you to send control messages in your aggregation flows, this is possible. To send control messages in a message flow created in WebSphere Message Broker Version 6.0, see “Configuring message flows to send control messages” and “Configuring a broker environment to send control messages” on page 398 in this topic.

If you created message flows in Version 5.0 and configured them to use control messages, and wish to continue using control messages, see “Configuring a broker environment to send control messages” on page 398 in this topic. **Connections between the AggregateControl and AggregateReply nodes created in earlier versions of the product will be ignored in Version 6.0 unless you complete this task.**

For a working example of aggregation (without the use of control messages), see the Airline Reservations sample.

Configuring message flows to send control messages

To configure message flows to send control messages from an AggregateControl node to an AggregateReply node:

1. Switch to the Broker Application Development perspective.
2. If you have created the fan-out and fan-in flows in a single message flow:
 - a. Open the aggregation message flow.
 - b. Connect the control terminal of the AggregateControl node to the control terminal of the AggregateReply node to make the association.

This is referred to as a direct connection between the two aggregation nodes.

3. If you have created separate fan-out and fan-in message flows:
 - a. Open the fan-out message flow.

- b. Configure the AggregateControl node as described in “Creating the aggregation fan-out flow” on page 385.
- c. At this stage you can configure a Compute node that creates a valid output message containing the control message. For example, if you want to pass the control message to an MQOutput node, configure the Compute node to add an MQMD to the message and complete the required fields in that header. For example, you can code the following ESQL:

```
SET OutputRoot.MQMD.StrucId = MQMD_STRUC_ID;
SET OutputRoot.MQMD.Version = MQMD_CURRENT_VERSION;
SET OutputRoot.MQMD.Format = MQMD_STRING;
```

- d. Configure an output node that represents the intermediate destination for the control message. For example, if you want to send the control message to an intermediate WebSphere MQ queue, include an MQOutput node and identify the target queue in the Basic properties *Queue Manager Name* and *Queue Name*.
- e. Connect the control terminal of the AggregateControl node to the in terminal of the Compute node, and the out terminal of the Compute node to the in terminal of the output node that represents the intermediate destination for the control message.
- f. Open the fan-in message flow.
- g. Configure one input node to receive the reply messages, and as described in “Creating the aggregation fan-in flow” on page 389. This input node will also receive the control information from the AggregateControl node. For example, set the Basic property *Queue Name* of the MQInput node to receive the response and control message from an intermediate WebSphere MQ queue.
- h. Add a filter node to your fan-in flow after the input node and before the AggregateReply node as described in “Avoiding thread starvation on fan-in flows” on page 399.
- i. Connect the out terminal of the input node to the in terminal of the Filter node.
- j. Connect the out terminals of the Filter node to the control terminal and in terminal of the AggregateReply node.

This is referred to as an indirect connection between the two aggregation nodes.

Configuring a broker environment to send control messages

By default, in WebSphere Message Broker Version 6.0 any connections from the control terminal of the AggregateRequest node to the AggregateReply node are ignored. For these connections to be active, create the MQSI_AGGR_COMPAT_MODE environment variable in the broker’s environment. By default the environment variable does not exist. The existence of the environment variable means connections from the AggregateControl node are active, regardless of the value the environment variable is set to.

When the MQSI_AGGR_COMPAT_MODE environment variable has not been created, the default behavior for aggregation fan-out flows will be used. If the control terminal of the AggregateControl node is connected, either directly or indirectly to the in terminal of the AggregateReply node, this connection will be ignored and no control message will be sent.

If the `MQSI_AGGR_COMPAT_MODE` environment variable is created, the default behavior for aggregation fan-out flows will **not** be used, allowing you to send control messages from the `AggregateControl` node to the `AggregateReply` node. If the Control terminal of the `AggregateControl` node is connected, either directly or indirectly to the In terminal of the `AggregateReply` node, as described in “Creating the aggregation fan-out flow” on page 385, this connection will be recognized and a control message will be sent. Please note that this is not the optimal configuration and may decrease performance.

To create the `MQSI_AGGR_COMPAT_MODE` variable to allow connections between `AggregateControl` and `AggregateReply` nodes to be recognized:

- On Windows:
 1. Open System Properties: **Start** → **Control Panel** → **System**
 2. Click on the **Advanced** tab.
 3. Click **Environment Variables**.
 4. Under the **System variables** pane, click **New**.
 5. Under **Variable name** type `MQSI_AGGR_COMPAT_MODE`. If you want you can fill in the **Variable value**, otherwise leave it blank.
 6. For the environment variable to take effect you must restart the computer.
- On Linux, UNIX and z/OS:
 1. Edit the profile of the broker userid and include the following: `export MQSI_AGGR_COMPAT_MODE=`
 2. Reload the profile.
 3. Restart the broker.

Avoiding thread starvation on fan-in flows

This topic only applies if the Control terminal of the `Aggregate Control` node in your fan-out flow is connected to output control messages to a queue. By not connecting the Control terminal you can overcome the issues discussed in this section. For further information about connecting the Control terminal of the `AggregateControl` see “Using control messages in aggregation flows” on page 397.

The `Aggregate Reply` node has two input terminals: In and Control. If you use both of these terminals, remembering that the use of the Control terminal is optional, the most efficient way to supply data to the `Aggregate Reply` node is to have a single `MQInput` node for the fan-in flow followed by a `Filter` node. The `Filter` node is used to route an incoming message to the In or Control terminals of the `Aggregate Reply` node as appropriate.

Use a single `MQInput` followed by a `Filter` node instead of two `MQInput` nodes in the message flow: one for the In terminal and one for the Control terminal. You should use a single `MQInput` node because there is no means of specifying how any additional threads (made available by the use of additional instances) should be distributed between the two `MQInput` nodes. Traffic on the `AggregateReply` node’s In terminal is likely to be higher, therefore it is useful to have more threads running in its input node, it is not possible to configure this using two `MQInput` nodes. It is therefore possible for the node to be starved of threads, backing up reply messages and stalling the aggregation mechanism.

Use an `ESQL` module similar to the one shown below in your `Filter` node, to ensure that the messages are routed to the appropriate terminal of the `AggregateReply` node:

```

CREATE FILTER MODULE FanIn_Filter
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    IF Root.XML.ComIbmAggregateControlNode IS NULL THEN
        RETURN TRUE; -- wired to In
    ELSE
        RETURN FALSE; -- wired to Control
    END IF;
END;
END MODULE;

```

Note: If it not possible for you to configure your fan-in flow as described above, you can force the MQInput node that is reading control messages to run single-threaded. Do this by configuring the MQInput node. Set the Order Mode property on the advanced properties panel to be By Queue Order and by selecting Logical Order. This frees up all of the configured additional instances to be used by the other MQInput node. Please note, because the performance of the first MQInput node will be severely limited this configuration should only be used if you have no alternatives.

Handling exceptions in aggregation flows

When you use aggregation flows, you might find that exceptions occur. This topic tells you how to deal with them.

Before you start:

To complete this task, you must have completed the following tasks:

- “Creating the aggregation fan-out flow” on page 385
- “Creating the aggregation fan-in flow” on page 389

Dealing with exceptions

If an error is detected downstream of an AggregateReply node, the broker throws an exception. Another node in the message flow might also throw an exception using the ESQL THROW statement. In either case, when an exception is thrown, it is caught in one of two places:

- The input node on which the replies arrive
- The AggregateReply node

The following table lists events and what happens to an exception thrown downstream of the AggregateReply node.

Event	Message propagated	Output terminal	Exception caught at
An expected reply arrives at the input node and is passed to the In terminal of the AggregateReply node. It is the last reply needed to make an aggregation complete.	Aggregated reply message containing all the replies	Out	Input node
An unexpected reply arrives at the input node and is passed to the AggregateReply node. It is not recognized as a valid reply, and the Unknown Message Timeout property is set to 0.	Message received	Unknown	Input node

Event	Message propagated	Output terminal	Exception caught at
A timeout occurs because all the replies for an aggregation have not yet arrived.	Aggregated reply message containing all the replies that have been received	Timeout	AggregateReply node
An unknown timeout occurs because a retained message was not identified as a valid reply.	Retained message	Unknown	AggregateReply node
An aggregation is discovered to be complete at some time other than when the last reply arrived.	Aggregated reply message containing all the replies	Out	AggregateReply node

If you want to handle errors that occur in aggregation flows, you must catch these exceptions at all instances of each of these nodes in the message flow. To do this:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with.
3. If you want to handle these exceptions yourself, connect the catch terminal of each input and AggregateReply node to a sequence of nodes that handles the error that has occurred.

If you want a unified approach to error handling, attach the catch terminals of all these nodes to a single sequence of nodes, or create a subflow that handles errors in a single consistent manner and attach that subflow to each catch terminal.

4. If you want the broker to handle these exceptions using default error handling, do not connect the catch terminals of these nodes.

If you connect the catch terminal of the AggregateReply node, and want to output the message propagated through this terminal to a destination from which it can be retrieved for later processing, you must include a Compute node in the catch flow to provide any transport-specific processing. For example, you must add an MQMD header if you want to put the message to a WebSphere MQ queue from an MQOutput node.

The ESQL example below shows you how you can add an MQMD header and pass on the replies received by the AggregateReply node:

```
-- Add MQMD
SET OutputRoot.MQMD.Version = 2;
.
-- Include consolidated replies in the output message
SET OutputRoot.XML.Data.Parsed = InputRoot.ComIbmAggregateReplyBody;
.
```

If you want to propagate the information about the exception in the output message, you must also set the *Compute mode* property of the Compute node to a value that includes Exception.

Related concepts

“Message flows overview” on page 3

“Message flow aggregation” on page 90

Related tasks

“Configuring aggregation flows” on page 384

“Creating the aggregation fan-out flow” on page 385

“Creating the aggregation fan-in flow” on page 389
“Associating fan-out and fan-in aggregation flows” on page 393
“Setting timeouts for aggregation” on page 394
“Using multiple AggregateControl nodes” on page 395
“Handling errors in message flows” on page 111
“Designing a message flow” on page 59
“Creating a message flow” on page 126
“Defining message flow content” on page 135

Related reference

“AggregateControl node” on page 483
“AggregateReply node” on page 485
“AggregateRequest node” on page 488

Exceptions when dealing with unknown and timeout messages

When timeout messages or unknown messages from unknown timeout processing are produced from an AggregateReply node they originate from a internal queue and not from a MQInput node. This effects how the error handling should be performed.

If a message sent down the timeout thread causes an exception, the message rolls back to the AggregateReply node and is sent to the catch terminal. If this terminal is either unattached or an exception occurs while processing the message, the timeout is rolled back onto the internal queue and is reprocessed. Potentially, this will lead to an infinite loop which can only be stopped either by removing the timeout message from the internal queue (not recommended), or by deploying a version of the messages flow that fixes the problem.

To avoid this infinite loop take the following actions.

- Connect the catch terminal up to a error handling set of nodes.
- Ensure the error handling nodes cannot throw an exception by ensuring that the perform very simple operations, for example, converting the message to a blob and then writing it to a queue, or add in extra TryCatch nodes.

Note: The failure terminal is currently not used and messages will never be passed to this terminal.

Configuring timeout flows

Use the TimeoutControl and TimeoutNotification nodes in message flows to process timeout requests or to generate timeout notifications at specified intervals.

The following scenarios show how these nodes can be used in a message flow:

- “Sending a message after a timed interval” on page 404
- “Sending a message multiple times after a specified start time” on page 405
- “Automatically generating messages to drive a flow” on page 406

Timeout request message

The XML format of a timeout request message is specified below. Any other format that is supported by an installed parser can be used instead of XML.

```

<TimeoutRequest>
  <Action>SET | CANCEL</Action>
  <Identifier>String (any alphanumeric string)</Identifier>
  <StartDate>String (TODAY | yyyy-mm-dd)</StartDate>
  <StartTime>String (NOW | hh:mm:ss)</StartTime>
  <Interval>Integer (seconds)</Interval>
  <Count>Integer (greater than 0 or -1)</Count>
  <IgnoreMissed>TRUE | FALSE</IgnoreMissed>
  <AllowOverwrite>TRUE | FALSE</AllowOverwrite>
</TimeoutRequest>

```

Action

This element must be set to either SET or CANCEL. It is an error to omit this element or to set a different value. If CANCEL is set, the only other element that is required is the Identifier, which must match the Identifier of the TimeoutRequest that is to be cancelled.

Identifier

This can be any alphanumeric string. It is an error to omit this element.

StartDate

This element must be set to TODAY or to a date specified in yyyy-mm-dd format. The default value is TODAY.

StartTime

This element must be set to NOW or to a time specified in hh:mm:ss format. The default value is NOW. StartTime is assumed to be the broker's local time.

Interval

This is an integer that specifies the number of seconds between propagations of the message. The default value is 0.

Count

This must be an integer that is either greater than 0 or is -1 (which specifies a timeout request that never expires). The default value is 1.

IgnoreMissed

This controls whether timeouts that occur while either the broker or the timeout notification flow is stopped, are processed the next time that the broker or timeout notification flow is started. The default value is TRUE which means that missed timeouts are ignored by the TimeoutNotification node when the broker or message flow is started. If this value is set to FALSE then the missed timeouts are all immediately processed by the Timeout Notification node when the flow is started.

Note that the *Request Persistence* property of the TimeoutControl node must be set to Yes or Automatic (with the originating request message being persistent) for the stored timeouts to persist beyond the restart of the broker or the timeout notification flow.

AllowOverwrite

This controls whether subsequent timeout requests with a matching *Identifier* can overwrite this timeout request. The default value is TRUE.

Note: A predefined schema definition of the timeout request is provided in the broker toolkit. Take the following steps to review the definition or define it within a message set:

1. Create or select a message set project that contains the message set.
2. Create a new message definition file.
3. Select IBM supplied message.

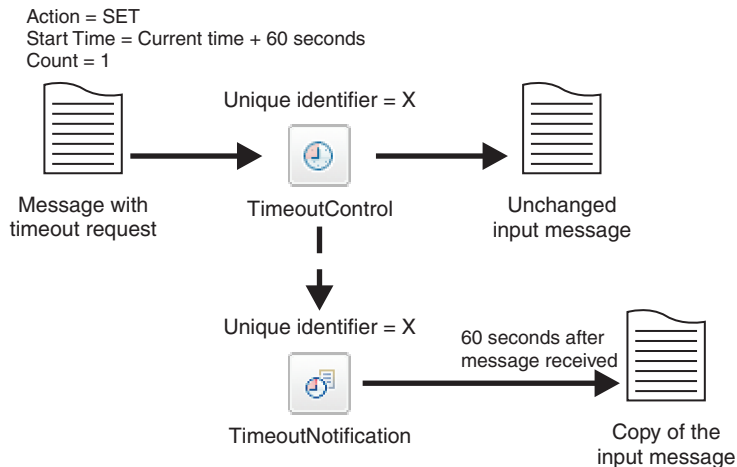
4. Select next, expand the tree, and select 6.0.0\ibm\nodes\timeout\timeoutrequest.xsd.

Sending a message after a timed interval

Aim

Use TimeoutControl and TimeoutNotification nodes to send a message into a message flow 60 seconds after the message is received.

Description of the flow



The diagram shows the path of a message that contains a timeout request through a TimeoutControl node. A TimeoutNotification node with an identifier matching the TimeoutControl node then processes the timeout request. The diagram also shows the message that the TimeoutNotification node produces after processing the timeout request.

The message comes into the TimeoutControl node with the following values set in the timeout request section of the message:

Action set to *SET*
Start Time set to *current time + 60*
Count set to *1*

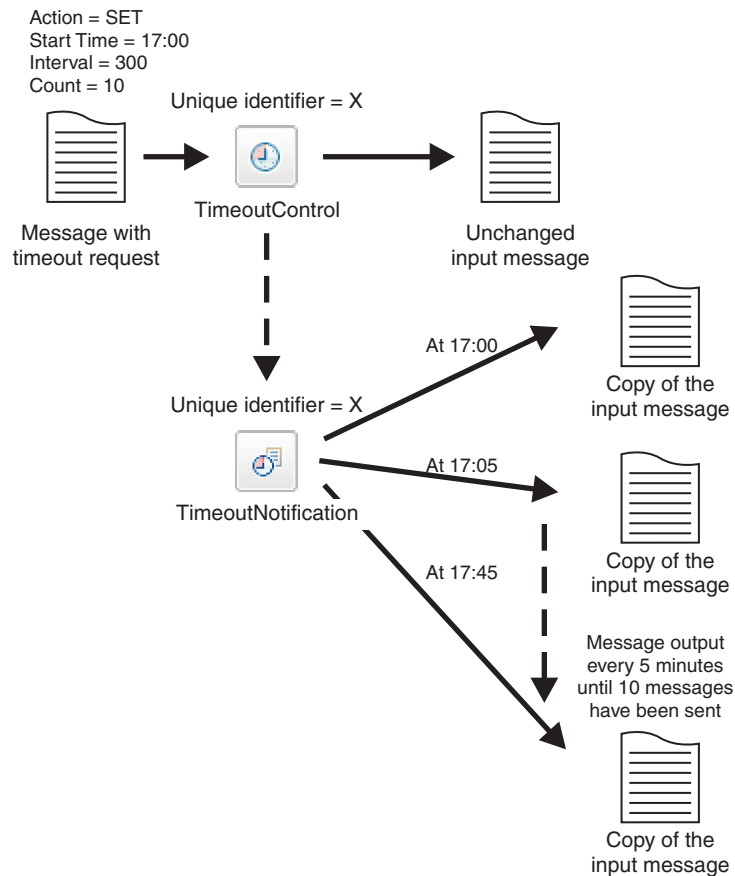
The TimeoutControl node validates the timeout request; default values are assumed for properties that are not explicitly defined. The original message is then sent on to the next node in the message flow. If the request is valid, the TimeoutNotification node with the same Unique identifier as the TimeoutControl node propagates a copy of the message to the message flow 60 seconds after the message was received.

See the Timeout Processing sample sample for further details on constructing this type of message flow.

Sending a message multiple times after a specified start time Aim

Use TimeoutControl and TimeoutNotification nodes to send a message into a message flow at 17:00 hours and then send the message again every 5 minutes until the message has been sent 10 times.

Description of the flow



The diagram shows the path of a message that contains a timeout request through a TimeoutControl node. A TimeoutNotification node with an identifier matching the TimeoutControl node then processes the timeout request. The diagram also shows the message that the TimeoutNotification node produces after processing the timeout request.

The message comes into the TimeoutControl node with the following values set in the timeout request section of the message:

Action set to SET
Start Time set to 17:00
Interval set to 300
Count set to 10

The TimeoutControl node validates the timeout request; default values are assumed for properties that are not explicitly defined. The original message is then sent on to the next node in the message flow. If the request is valid, the TimeoutNotification node with the same Unique identifier as the TimeoutControl

node propagates a copy of the message to the message flow at 17:00. The message is sent again after an interval of 300 seconds, at 17:05. and every 300 seconds until the message has been sent 10 times, as the *Count* value in the timeout request specifies.

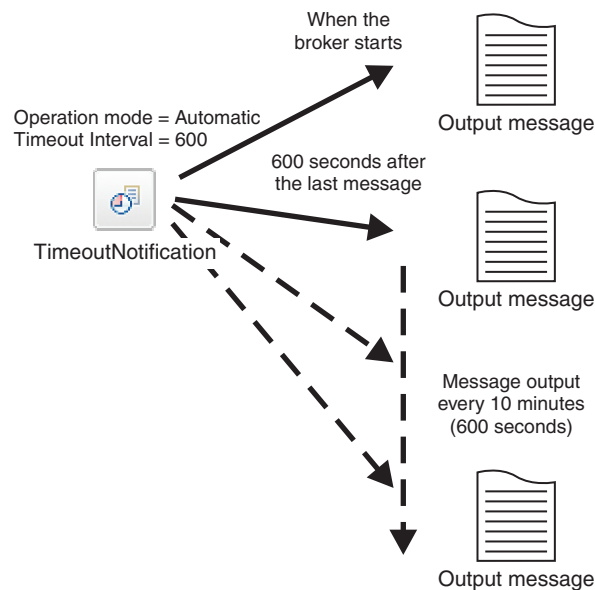
See the Timeout Processing sample sample for further details on constructing this type of message flow.

Automatically generating messages to drive a flow

Aim

Use the TimeoutNotification node to automatically send a message into a message flow every 10 minutes.

Description of the flow



The diagram shows a TimeoutNotification node automatically generating messages and propagating them every 10 minutes. To get the TimeoutNotification node to automatically generate messages, set the Operation Mode property of the node to automatic and specify a value for the Timeout Interval property. In this example the TimeoutNotification node has the following properties:

- Operation Mode set to automatic
- Timeout Interval set to 600

When the broker has started, the TimeoutNotification node sends a message every 10 minutes (600 seconds). This message contains only a properties folder and a LocalEnvironment folder. A Compute node can then process this message to create a more meaningful message.

See the Timeout Processing sample sample for further details on constructing this type of message flow.

Using an MQGet node in a request-response flow

For details of the processing done within the MQGet node to achieve the above, see MQGet node message processing. For details on constructing this type of flow, see the sample Coordinated Request Reply sample

Introduction

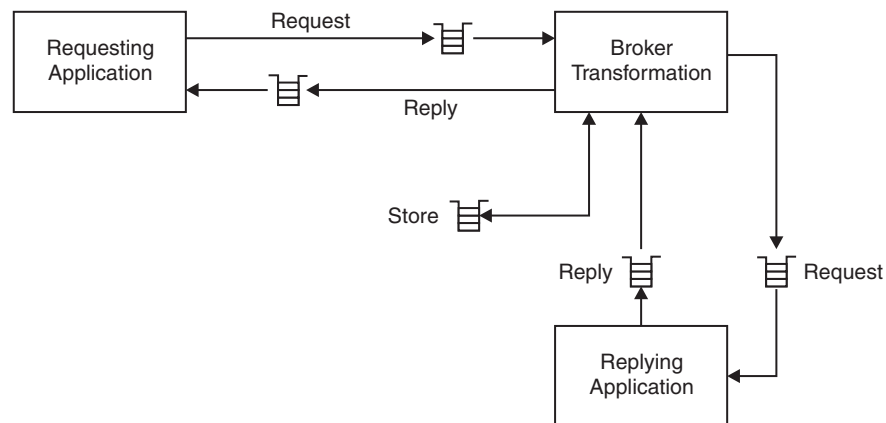
This page is an introduction to the use of an MQGet node in a request-response flow, and describes how the node processes the input messages (according to the LocalEnvironment and input parameters you set) to construct the output messages.

A request-response flow between two applications allows one of the applications to request messages from the other. This type of flow is illustrated in the following diagram:



In the diagram, the Requesting Application places a message into the "Request" input queue of the Replying Application. The Replying Application then processes the message and sends a reply to the "Reply" queue specified in the original message from the Requesting Application.

If one of these applications is to be replaced or enhanced without making changes to the other, then messages will need to be transformed between the two applications. To achieve this, the Broker can be inserted between them, as shown in the following diagram:



Now a queue alias, or a similar device, is configured so that the Requesting Application places its request message into the input queue of the "Coordinated Request Reply" Broker message flow. In the previous example this message was placed directly into the input queue of the Replying Application. The "Coordinated Request Reply" flow transforms the message to a format suitable for the Replying Application, and passes it on to its input queue. It also saves the original

Requesting Application's reply-to queue details, and restores these to the reply message it receives from the Replying Application so it can be posted back correctly to the Requesting Application.

An MQGet node can be placed anywhere in a flow and receives on its input terminal an input tree propagated from the preceding node. It then uses the message retrieved from the configured WebSphere MQ queue to build a result tree. Finally, it uses the input tree and the result tree to build an output tree that is then propagated to its output, warning or failure terminal depending on the configuration of the node and the result of the Get operation.

For more details on constructing a flow, see the sample: Coordinated Request Reply sample.

How the LocalEnvironment is used

The LocalEnvironment propagated from the preceding node is read and updated by the MQGet node.

- The MQGMO structure will be read from `${inputMQParmsLocation}.MQGMO.*`
- MQ completion and reason codes will be placed in `${outputMQParmsLocation}.CC` and `.RC`
- If an MQGMO tree exists in the local environment, it will be updated with the values used by the node and propagated downstream
- If `${inputMQParmsLocation}.MQMD` exists, then the MQMD passed to the MQGET call itself (containing the values specified in the input message or generated by the node) will be placed in that location, deleting anything already in that location.

`${inputMQParmsLocation}` is the value set in the MQGet Node Property *Input MQ Parameters Location* on the **Request Properties** tab.

`${outputMQParmsLocation}` is the value set in the MQGet Node Property *Output MQ Parameters Location* on the **Result Properties** tab.

For details of these properties, see "MQGet node" on page 584.

In summary:

`${inputMQParmsLocation}`

- *QueueName*: Optional override for MQGet Node Property *Queue Name*.
- *InitialBufferSize*: Optional override for MQGet Node Property *Initial Buffer Size*.
- *MQGMO.**: Optional MQ Get message options the MQGet Node will use.

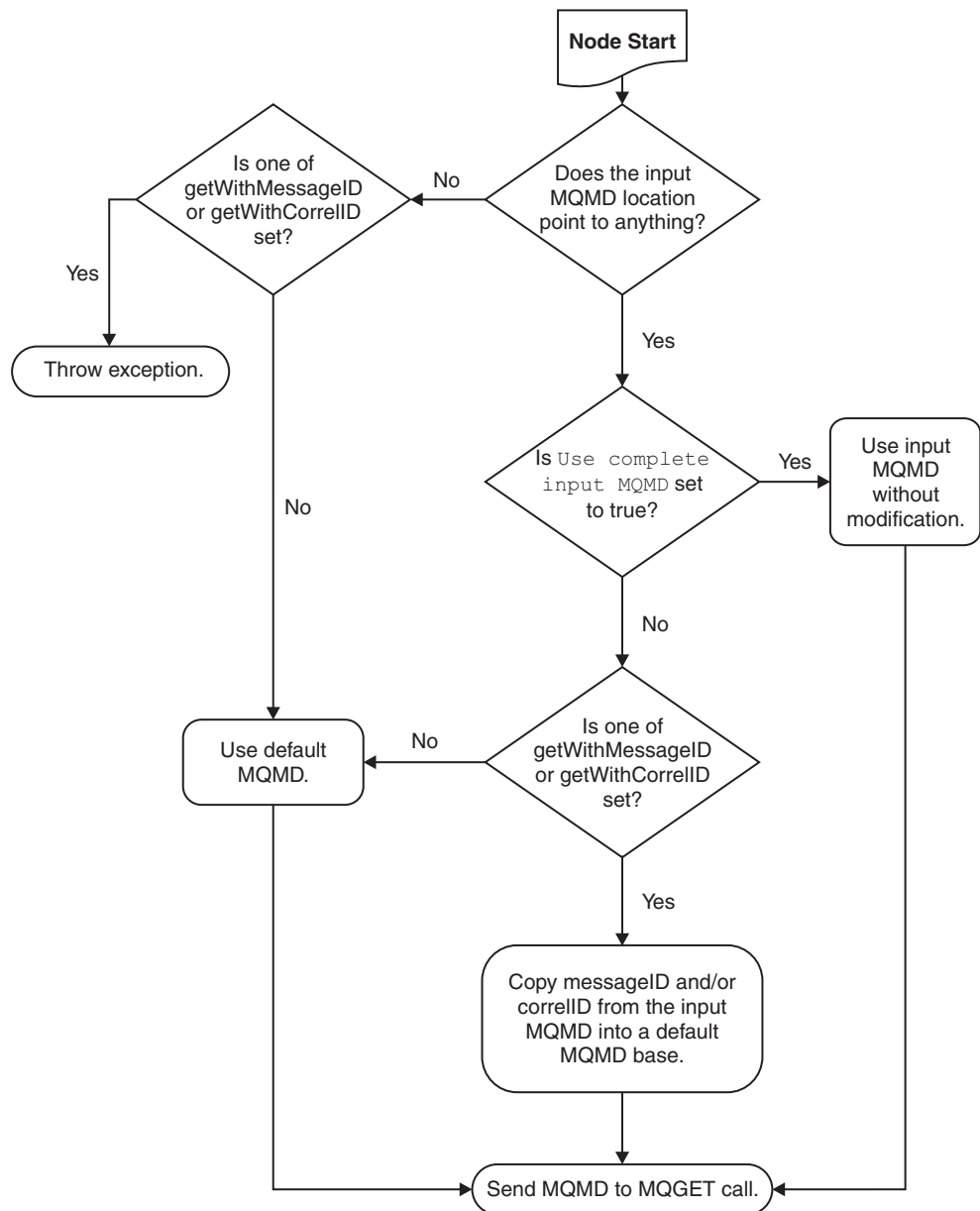
`${outputMQParmsLocation}`

- *CC*: MQ Get call completion code.
- *RC*: MQ Get call result code.
- *MQGMO.**: MQ Get message options use if present in `${inputMQParmsLocation}`.
- *MQMD*: MQ Message Descriptor for received messages.

How the MQMD for the MQGet call is constructed

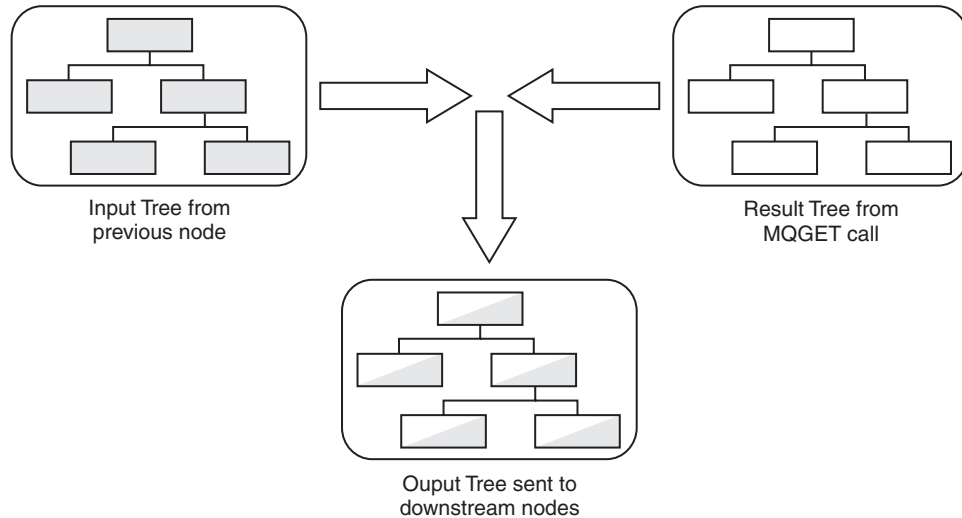
- If you do not supply an input MQMD then a default MQMD, as described in the *WebSphere MQ Application Programming Reference*, is used.
- If you do supply an input MQMD, then it is used in one of the two following ways:
 - If the attribute **Use complete input MQMD** is set, the input MQMD is used in its entirety.
 - If the attribute **Use complete input MQMD** is not set, a default MQMD is prepared, then from the input MQMD, if the check boxes **messageID** or **correlID** are set, then the respective IDs are copied into it.

The following diagram shows in a little more detail how the MQGet node constructs the MQMD to be used on the call to WebSphere MQ:



How the output message tree is constructed

The following diagram outlines how the output message tree is constructed by combining the input tree from the previous node with the result tree from the MQGet call:



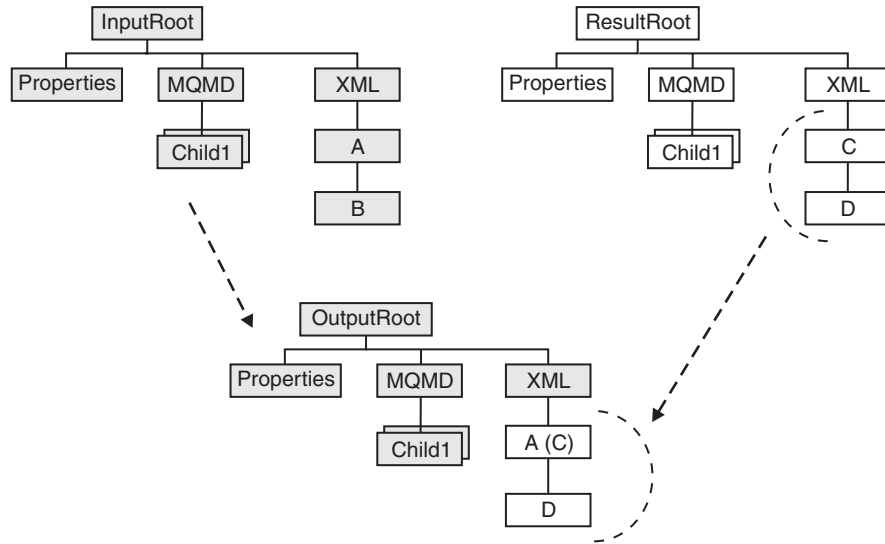
Here is an example, where the MQGet Node properties are configured as listed:

```
copyMessage  
  copyEntireMessage  
generateMode  
  message  
outputDataLocation  
  OutputRoot.XML.A  
resultDataLocation  
  ResultRoot.XML.C
```

In this example the Output tree is constructed according to the following sequence:

1. The whole of the Input tree is copied to the Output tree, including the XML branch with child A and A's child B.
2. From the Result tree, the XML branch's child C and C's child D are put into the Output tree at position OutputRoot.XML.A. Any previous content of A (values and children) is lost, and replaced with C's content, including all values and children it has, in this case child D.
3. The position in the Output tree remains named A.

The following diagram illustrates this visually:



Message tree examples

Here are some examples of how message trees are constructed according to the rules outlined above.

Table 2. An example Input, LocalEnvironment and MQGet

With a message assembly like this:	The message that MQGet returns is:
InputRoot MQMD {input message mqmd} MQRFH2 {input message mqrh2} XMLNS {input message body}	ResultRoot MQMD {result message mqmd} MQMD {result message mqmd} XML {result message body}
InputLocalEnvironment MQ GET MQGMO MatchOptions = MQMO_MATCH_CORREL_ID MQMD (with no children)	
MyData MQMD {input mqmd} (with CorrelID = {correct Correlation ID as binary})	

Table 3. Example resulting output message trees, according to the MQGet Node properties settings detailed.

With the following settings:	The resulting output message assembly is:
<p>inputMQMDLocation InputLocalEnvironment.MyData.MQMD</p> <p>copyMessage copyEntireMessage</p> <p>copyLocalEnv copyEntireLocalEnvironment</p> <p>generateMode messageAndLocalEnvironment</p> <p>outputDataLocation InputLocalEnvironment.MyData.ReturnedMessage</p>	<p>OutputRoot</p> <p>MQMD {input message mqmd}</p> <p>MQRFH2 {input message mqrfh2}</p> <p>XMLNS {input message body}</p> <p>OutputLocalEnvironment</p> <p>MQ</p> <p>GET</p> <p>MQGMO {mqgmo used for get}</p> <p>MQMD {mqmd used for get}</p> <p>CC = 0</p> <p>RC = 0</p> <p>MyData</p> <p>MQMD {input mqmd} (with CorrelID = {correct Correlation ID as binary})</p> <p>Returned message</p> <p>MQMD {result message mqmd}</p> <p>MQRFH2 {result message mqrfh2}</p> <p>XML {result message body}</p>

Table 3. Example resulting output message trees, according to the MQGet Node properties settings detailed. (continued)

With the following settings:	The resulting output message assembly is:
<p>resultDataLocation ResultRoot.XML</p>	<p>OutputRoot</p> <ul style="list-style-type: none"> MQMD {input message mqmd} MQRFH2 {input message mqrfh2} XMLNS {input message body} <p>OutputLocalEnvironment</p> <ul style="list-style-type: none"> MQ GET MQGMO {mqgmo used for get} MQMD {mqmd used for get} CC = 0 RC = 0 <p>MyData</p> <ul style="list-style-type: none"> MQMD {input mqmd} (with CorrelID = {correct Correlation ID as binary}) <p>Returned message (with any attributes and value from ResultRoot.XML) {result message body}</p> <p>This tree is effectively the result of doing an assignment from \${resultDataLocation} to \${outputDataLocation}. The value of the source element is copied, as are all children including attributes.</p>

Table 3. Example resulting output message trees, according to the MQGet Node properties settings detailed. (continued)

With the following settings:	The resulting output message assembly is:
<p>copyLocalEnv none</p>	<p>OutputRoot</p> <ul style="list-style-type: none"> MQMD {input message mqmd} MQRFH2 {input message mqrfh2} XMLNS {input message body} <p>OutputLocalEnvironment</p> <ul style="list-style-type: none"> MQ <ul style="list-style-type: none"> GET <ul style="list-style-type: none"> MQGMO {mqgmo used for get} MQMD {mqmd used for get} CC = 0 RC = 0 MyData <ul style="list-style-type: none"> Returned message (with any attributes and value from ResultRoot.XML) {result message body} <p>This tree has the MQMD used for the get in the OutputLocalEnvironment because the input MQ parameters location had an MQMD element under it. Even though the input tree is not copied, the presence of the MQMD element causes the MQMD used for the get to be placed in the output tree.</p>

Table 3. Example resulting output message trees, according to the MQGet Node properties settings detailed. (continued)

With the following settings:	The resulting output message assembly is:
<p>outputDataLocation <blank></p> <p>copyLocalEnv copyEntireLocalEnvironment</p>	<p>OutputRoot</p> <ul style="list-style-type: none"> MQMD {result message mqmd} MQRFH2 {result message mqrh2} XMLNS {result message body} <p>OutputLocalEnvironment</p> <ul style="list-style-type: none"> MQ <ul style="list-style-type: none"> GET <ul style="list-style-type: none"> MQGMO {mqgmo used for get} MQMD {mqmd used for get} CC = 0 RC = 0 MyData <ul style="list-style-type: none"> MQMD {input mqmd} (with CorrelID = {correct Correlation ID as binary}) <p>The setting of copyMessage in this case makes no difference to the eventual output tree.</p>

Part 2. Deploying

Deploying	419
Deployment overview	419
Deployment environments	420
Types of deployment	421
Message flow application deployment	422
Broker configuration deployment	426
Publish/subscribe topology deployment	426
Publish/subscribe topics hierarchy deployment	427
Cancel deployment	428
Deploying a message flow application	429
Creating a server project	430
Creating a broker archive	431
Adding files to a broker archive	432
Deploying a broker archive file	434
Configuring a message flow at deployment time using UDPs	436
Deploying a broker configuration	436
Using the Message Brokers Toolkit	436
Using the mqsideploy command	437
Using the Configuration Manager Proxy API	437
Deploying a publish/subscribe topology	438
Using the Message Brokers Toolkit	438
Using the mqsideploy command	438
Using the Configuration Manager Proxy API	439
Deploying a publish/subscribe topics hierarchy	439
Using the Message Brokers Toolkit	440
Using the mqsideploy command	440
Using the Configuration Manager Proxy API	441
Checking the results of deployment	441
Using the Message Brokers Toolkit	441
Using the mqsideploy command	442
Using the Configuration Manager Proxy API	442
Canceling a deployment that is in progress	443
Using the Message Brokers Toolkit	444
Using the mqsideploy command	444
Using the Configuration Manager Proxy API	444
Renaming objects that are deployed to execution groups	445
Removing a deployed object from an execution group	445
Using the Message Brokers Toolkit	445
Using the mqsideploy command	446
Using the Configuration Manager Proxy API	446

Deploying

This section provides information about deploying to execution groups on brokers.

It starts with an overview of deployment, which includes descriptions of the different mechanisms you can use to deploy. The section also contains separate topics describing the different types of deployment:

- **Deployment overview**
 - Message flow application deployment
 - Broker archive file
 - Configurable properties
 - Object version and keyword
 - Broker configuration deployment
 - Publish/subscribe topology deployment
 - Publish/subscribe topics hierarchy deployment
 - Cancel deployment

It then describes in detail the steps needed to deploy message flow applications:

- **Deploying a message flow application**
 - Creating a server project
 - Creating a broker archive
 - Adding files to a broker archive
 - Editing a broker archive file manually
 - Editing configurable properties
 - Deploying a broker archive
 - Using UDPs to configure a message flow at deployment time

It also explains how to perform other types of deployment:

- Deploying a broker configuration
- Deploying a publish/subscribe topology
- Deploying a publish/subscribe topics hierarchy

It finally deals with a number of other deployment-related tasks:

- Checking the results of deployment
- Canceling a deployment
- Renaming a deployed object
- Removing a deployed object

Deployment overview

Deployment is the process of transferring data to an execution group on a broker so that it can take effect in the broker domain. For deploying message flows and associated resources, the data is packaged in a broker archive (bar) file before being sent to the Configuration Manager, from where it is unpackaged and distributed appropriately.

This topic describes the three environments from which you can perform a deployment, and then introduces a number of different types of deployment that you might need to use:

- “Deployment environments”
- “Types of deployment” on page 421

You will also find that most types of deployment can, typically, be configured to perform in one of two ways:

- complete deployment; in which *everything* is deployed (or re-deployed) to the whole domain
- delta or incremental deployment; made either to just update information or to deploy to selected brokers within the domain, depending on the type of deployment

After reading this conceptual overview, find detailed instructions for particular tasks in the subsequent topics.

You can read a further discussion about deployment in the IBM Redbook, *WebSphere Message Broker Basics*.

Deployment environments

Depending on the environment in which you are working, you can generally choose one of three different ways to initiate a deployment:

Using the Message Brokers Toolkit

In the Broker Administration perspective of the workbench, the Domain Navigator view displays all the objects associated with a specific domain. For example, if you expand the Topology view, all the brokers in the domain are displayed; if you expand a Broker view, all the execution groups within that broker are displayed. From the Domain Navigator view you can deploy a topology to all the brokers in the domain, or you can deploy all the execution groups to a particular broker. You can also drag a broker archive (bar) file from the Resource Navigator view onto an execution group within the Domain Navigator view to deploy the contents of the bar.

You might typically use the workbench if you are working in a development environment or if you are new to WebSphere Message Broker.

Using the mqsideploy command

You can deploy from the command line using the mqsideploy command. On the command line, you must typically specify the connection details as well as parameters specific to the type of deployment you want to perform. Details are given in each topic describing the types of deployment.

You might typically use the mqsideploy command in a script when you are more familiar with WebSphere Message Broker.

WebSphere Message Broker provides two files to help you when writing your own scripts for managing broker deployment outside the workbench. These are:

- Initialization file mqsicfgutil.ini. This is a plain text file in the mqsideploy command’s working directory that contains configurable variables needed to connect to the Configuration Manager. For example:

```
hostname = localhost
queueManager = QMNAME
port = 1414
securityExit = test.myExit
```

If you do not explicitly specify any of this information as parameters on the mqsideploy command (as has been done in the examples in subsequent topics), the information is taken from the mqsicfgutil.ini file. Alternatively, use the -n parameter on the command to specify an XML-format .configmgr file that describes the connection parameters to the Configuration Manager.

- Batch file mqsideploy.bat. The parameters used with the mqsideploy command in WebSphere Message Broker Version 6.0 are not the same as those used in earlier versions of the command. On Windows platforms, use mqsideploy.bat if you want to use the same parameters as in previous versions.

Using the Configuration Manager Proxy API

You can control deployment from any Java program using the Configuration Manager Proxy API. You can also interrogate the responses from the broker and take appropriate action.

The Configuration Manager Proxy API also allows Java applications to control other objects in the domain, such as brokers, execution groups, publish/subscribe topologies, topics, subscriptions and the Configuration Manager and its event log. Because of this, you can use the Configuration Manager Proxy API to create and manipulate an entire domain programmatically.

Types of deployment

The other topics within this section describe what each type of deployment does, the situation in which each type should and should not be used.

- Broker configuration deployment

To deploy message flows, message sets and other deployable objects to an execution group, use:

- Message flow application deployment

This uses a broker archive file for deployment. You can set configurable properties for objects in the message flow.

In publish/subscribe scenarios, you can deploy topics and topologies:

- Topics hierarchy deployment
- Topology configuration deployment

You can also cancel a deployment.

This table lists appropriate ways of deploying for a number of common scenarios.

Scenario	Suggested deployment
Adding a broker to the domain (when not using publish/subscribe)	None required.
Connecting publish/subscribe brokers using connections or a collective	Delta topology deployment
Modifying the publish/subscribe topic hierarchy	Delta deployment of the topics hierarchy (The changed elements in the topic hierarchy are deployed to all brokers in the domain.)
Modifying the publish/subscribe topic hierarchy, after adding a new broker to the domain	Complete topics deployment (The entire topic hierarchy is deployed to all brokers in the domain. The new broker also receives the complete topic hierarchy.)

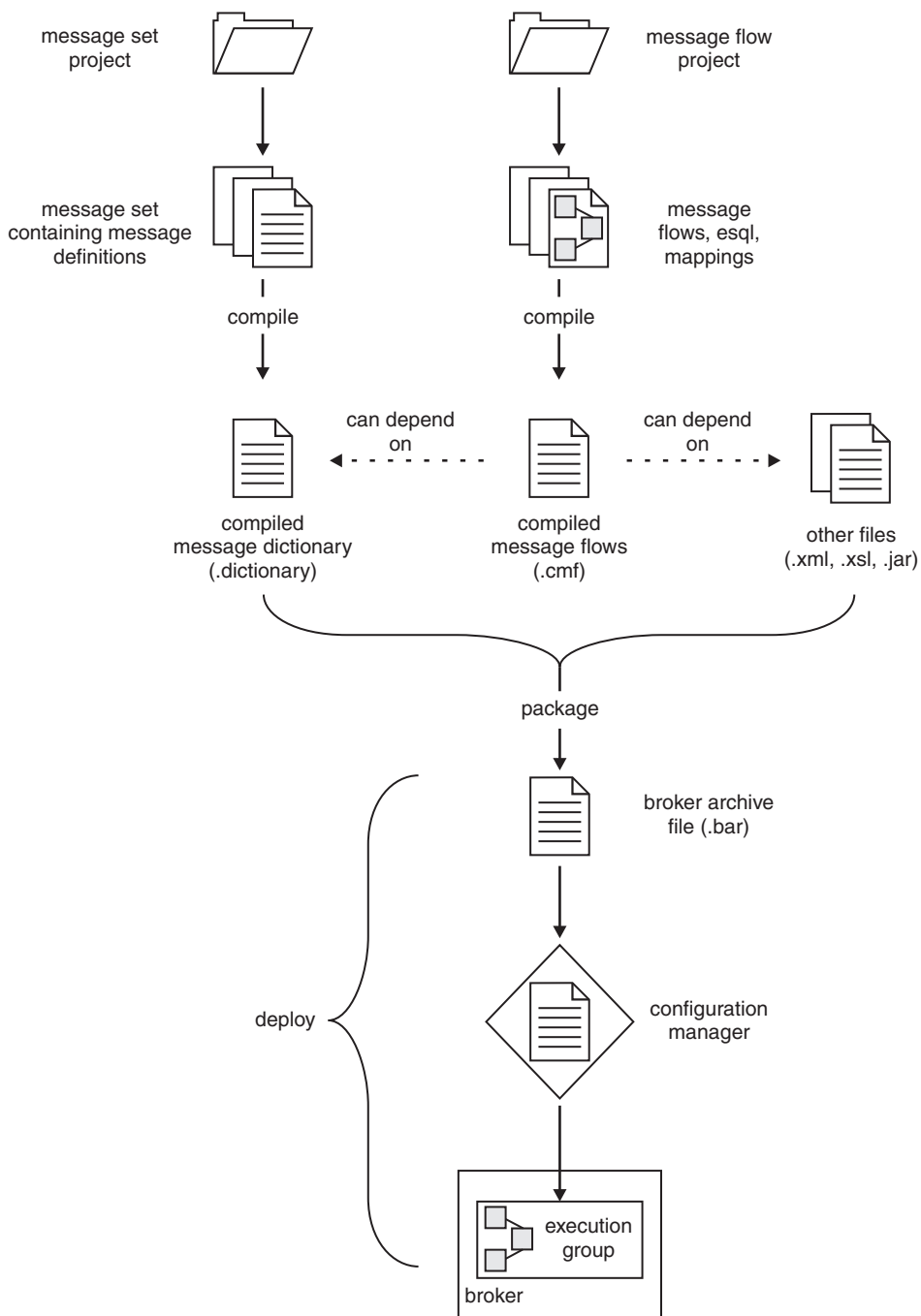
Scenario	Suggested deployment
Tidying up a broker's resources after removing it from the topology	If the broker is part of a publish/subscribe network, or if you are using the Message Brokers Toolkit, initiate a delta publish/subscribe topology deployment. Otherwise, no deployment is required.
Creating an execution group	Message flow application deployment using an incremental bar file deployment.
Deleting an execution group	None required.
If a broker is not responding to a deploy request	Ensure that the broker is running. If the broker is not running, cancel the broker deployment. You should only cancel a broker deployment if you are sure that the broker will never respond to the deploy request.

Message flow application deployment

You do not deploy a message flow application directly to an execution group. Instead, you package all the relevant resources into a broker archive (bar), which you then deploy. When you add files to the broker archive, they are automatically compiled as part of the process. JAR files that are required by JavaCompute nodes within message flows are added automatically from your Java project.

The broker archive itself is a compressed file which is sent to the Configuration Manager where its contents are extracted and distributed to execution groups. If an execution group has not been initialized on the broker (that is, if the broker has only just been created), then the execution group is created as part of the deployment.

This illustration shows the flow of events when you deploy a message flow application:



There are two ways of deploying a bar file:

- Incremental deployment, in which deployed files are added to the execution group. Files which already exist in the execution group are replaced with the new version.
- Complete deployment, in which files already deployed to the execution group are removed before the complete contents of the bar file is deployed. Thus, nothing is left in the execution group from any previous deployment.

Incremental bar file deployment

Incrementally deploying a bar file tells the Configuration Manager to extract the contents of the bar file and send it to an execution group.

- If a file in the bar file has the same name as an object that is already deployed to the execution group, the version that is already deployed is replaced with the version in the bar file.
- If a file in the bar file is of zero length and a file of that name has already been deployed to the execution group, then the deployed file is removed from the execution group.

When to use it

- If you want to incrementally deploy message flows, message sets or other deployable objects to an execution group.

When *not* to use it

- If you want to completely clear the contents of the execution group before the bar file is deployed. In this case, use a complete bar file deployment instead.

Complete bar file deployment

Completely deploying a bar file tells the Configuration Manager to extract the deployable content of the bar file and send it to an execution group, first removing any existing deployed contents of the execution group.

When to use it

- If you want to deploy message flows, message sets or other deployable objects to an execution group.

When *not* to use it

- If you want to merge the existing contents of the execution group with the contents of the bar file. In this case, use an incremental bar file deployment instead.

Broker archive

The unit of deployment to the broker is the *broker archive* or *bar* file.

The bar file is a zip-format file which can contain a number of different files:

- A .cmf file for each message flow. This is a compiled version of the message flow. You can have any number of these files within your bar file.
- A .dictionary file for each message set dictionary. You can have any number of these files within your bar file.
- A broker.xml file. This file is called the *broker deployment descriptor*. You can have only one of these files within your bar file. This file, in XML format, resides in the META-INF folder of the zip file and can be modified using a text editor or shell script.
- Any number of XML files(.xml) and style sheets (.xsl files) for use with the XMLTransformation node.
- Any number of JAR files for use with the JavaCompute node.
- As a zip-format archive, the broker archive file can also contain any additional files you need. For example, you might want to include Java source files for future reference.

To deploy XML, XSL, and JAR files inside a broker archive, the connected Configuration Manager and target broker must be Version 6.0 or later.

Broker archive - configurable properties

System objects defined in message flows can have configurable properties that you can update within the broker archive (bar) file before deployment. Configurable properties allow an administrator to update target-dependent attributes, such as queue names, queue manager names, and database connections, so that they can be customized for a new domain, for example to a test system, without needing to edit and rebuild the message flows, message mappings, or ESQL transformation programs. Any such properties that you define are contained within the deployment descriptor, META-INF/broker.xml.

Edit the configurable properties using the Broker Archive editor, or by manually editing the deployment descriptor using an external text editor or shell script. The deployment descriptor is parsed when the bar file is deployed.

Object version and keyword

This topic contains information about how to view the version and keyword information of deployable objects.

- “Displaying object version in the bar file editor”
- “Displaying version, deploy time, and keywords of deployed objects”

Displaying object version in the bar file editor

A column in the bar editor called *Version* displays the version tag for all objects that have a defined version. These are:

- .dictionary files
- .cmf files
- Embedded JAR files with a version defined in a META-INF/keywords.txt file

You cannot edit the *Version* column.

You can use the `mqsi readbar` command to list the keywords defined for each deployable file within a deployable archive file.

Displaying version, deploy time, and keywords of deployed objects

The Eclipse *Properties View* displays, for any deployed object:

- Version
- Deploy Time
- All defined keywords

For example, if you deploy a message flow with these literal strings:

- `$MQSI_VERSION=v1.0 MQSI$`
- `$MQSI Author=fred MQSI$`
- `$MQSI Subflow 1 Version=v1.3.2 MQSI$`

then the Properties View displays:

Deployment Time	Date and time of deployment
Modification Time	Date and time of modification
Version	v1.0
Author	fred

You are given a reason if the keyword information is not available. For example, if keyword resolution has not been enabled at deploy time, the Properties View displays the message Deployed with keyword search disabled. Also, if you deploy to a Configuration Manager that is an earlier version than Version 6.0, the properties view displays Keywords not available on this Configuration Manager.

Broker configuration deployment

A broker configuration deployment informs a broker of various configuration settings, including the list of execution groups, multicast and inter-broker settings.

When to use it

- If you have modified multicast or inter-broker settings in the Message Brokers Toolkit or in a Configuration Manager Proxy application.

When *not* to use it

- If you have used the `mqsichangeproperties` command to change publish/subscribe settings directly on the broker component. In this case, a broker configuration deployment overwrites any changes you have made to the settings.
- If you are adding execution groups. In this case, the first time you deploy a broker archive (`bar`) file, the execution group is automatically initialized.

Publish/subscribe topology deployment

Topology deployment is primarily only required when using publish/subscribe. It informs each broker in the domain of the brokers with which it can share publications and subscriptions.

There are two ways of deploying a topology configuration:

- Complete topology deployment, in which all brokers are told of their neighboring publish/subscribe brokers.
- Delta topology deployment, in which only changes to the publish/subscribe topology are deployed. Such changes are deployed only to those brokers whose neighbor lists have changed since the last successful topology deployment.

Complete topology deployment

Deploying a complete topology:

- Tells each broker in the domain the set of brokers with which it can share publish/subscribe information.
- Forces the Configuration Manager to re-subscribe to the broker's status topics, such as start and stop messages.

Note: Whatever type of deployment you perform, the Configuration Manager attempts to subscribe to the broker's status messages if it is the first deployment to the broker. But only deploying the complete topology initiates a re-subscription.

When to use it

- If the Configuration Manager is not correctly reporting whether it is in a stopped or started state.

- If you have moved a Configuration Manager from one queue manager to another.
- If a broker's publish/subscribe function has become inconsistent. An example of inconsistency would be, for example, if one broker is able to share publications with a second broker, but not the other way round.

When *not* to use it

- If you are adding brokers to the domain and you are not using publish/subscribe. That is, if you are not connecting brokers together so that they can share publications and subscriptions.
- If you are adding execution groups to a broker.
- If you have changed the publish/subscribe network. In this case, deploy a delta topology, if possible, so that you only deploy to those brokers affected by the changes you have made.
- If you have removed a broker from the domain.

Delta topology deployment

Deploying a delta topology sends updated publish/subscribe network information to any broker with a publish/subscribe configuration that the Configuration Manager determines not to be current.

When to use it

- If you have modified a publish/subscribe network.
- If you are using the workbench to remove a broker from the domain, the Configuration Manager automatically requests the broker component to stop message flows that are running and to tidy up any resources being used. If this operation fails for any reason, you can again request the broker to tidy up. Deploying a delta topology is the most convenient way.

When *not* to use it

- If you are adding brokers to the domain and you are not using publish/subscribe. That is, if you are not connecting brokers together so that they can share publications and subscriptions.
- If you are adding or removing execution groups.

Publish/subscribe topics hierarchy deployment

If you are using publish/subscribe, deploy the topics hierarchy in these situations:

- If you have modified the hierarchy of topics. The deployment communicates the new hierarchy to each broker.
- If you have added a broker to the domain and you want it to use the existing topics hierarchy. The deployment communicates the hierarchy to the new broker.

There are two ways of deploying a publish/subscribe topics hierarchy:

- Complete deployment, in which the complete topics hierarchy is sent to all the brokers in a domain.
- Delta deployment, in which changes to the topics hierarchy (made since the last topics deployment) are sent to all the brokers in a domain.

Complete topics deployment

A complete topics deployment sends the publish/subscribe topics hierarchy to all the brokers in a domain.

When to use it

- If you have made changes to the topics hierarchy and one of the brokers has an inconsistent view of the expected topics hierarchy.
- If you have added a new broker to the domain that uses the topics hierarchy.

When *not* to use it

- If you have changed the topics hierarchy. In this case, a delta topics deployment is usually sufficient.

Delta topics deployment

A delta topics deployment sends the publish/subscribe topics hierarchy to all the brokers in a domain.

When to use it

- If you have made changes to the topics hierarchy.

When *not* to use it

- If the topics hierarchy has not changed.

Cancel deployment

The Configuration Manager allows only one deployment to be in progress to each broker at any one time. If for some reason a broker does not respond to a deployment request, subsequent requests cannot reach the broker, because, to the Configuration Manager, a deployment is still in progress.

Canceling deployment tells the Configuration Manager to assume that a broker will never respond to an outstanding deploy. In most cases, the action does *not* remove any deployment messages that have been sent to the broker, nor does it alter the running configuration of the broker. (Thus, for any brokers that *have* successfully deployed a configuration, the deployed information remains on the broker.)

If a broker subsequently *does* provide a response to an outstanding deployment that has been canceled, the response is ignored by the Configuration Manager and there is then an inconsistency between what is running on the broker and the information that is provided by the Configuration Manager.

Because of this risk of inconsistency, only cancel a deployment as a last resort, and only if you are sure that a broker will never be able to process a previous deployment request. However, before canceling deployment, you can manually remove any outstanding deployment messages to ensure that they are not processed.

When canceling deployment across the domain, the locks for all outstanding deploys in the domain are removed. When canceling deployment for a specific broker, only the lock for that broker is removed.

Canceling deployment is the equivalent of the 'force deploy' action in previous versions, except that cancel does not redeploy domain information.

You can cancel a deployment:

- To all the brokers in a domain
- To a single broker

Cancel deployment to a domain

Canceling the deployment to a domain tells the Configuration Manager to assume that all brokers in the domain that have outstanding deployments will not respond. If a broker then does provide a response to an outstanding deploy that has been canceled, it will be ignored and there will be an inconsistency between what is running on the broker and the information that is provided by the Configuration Manager.

When applied to a domain, canceling deployment does not remove deployment messages that have been sent to the brokers, and does not change the brokers' running configuration.

When to use it

Cancel a domain deployment only if these two conditions are *both* met:

- You attempt a deployment and you receive error message BIP1510.
- Any of the brokers that have outstanding deployments are not responding.

When *not* to use it

- If a broker is simply taking a long time to respond to a deployment request. The broker might have been temporarily stopped, for example.
- If other users might be deploying to the domain at the same time.
- If only one broker is not responding, or a small number of brokers are not responding. In this case, cancel the broker deployment instead.

Cancel deployment to a broker

Canceling the deployment to a single broker tells the Configuration Manager to assume that a specific broker in the domain that has an outstanding deployment will not respond. If the broker then does provide a response to an outstanding deploy that has been canceled, it will be ignored and there will be an inconsistency between what is running on the broker and the information that is provided by the Configuration Manager.

When applied to an individual broker, canceling deployment causes the Configuration Manager to attempt to remove from the broker, deployment messages that have not yet been processed. This only succeeds if the broker and the Configuration Manager share the same queue manager and if the message has not already been processed by the broker.

When to use it

Cancel a domain deployment only if these two conditions are *both* met:

- You attempt a deployment and you receive error message BIP1510.
- The broker is not responding.

When *not* to use it

- If the broker is simply taking a long time to respond to a deployment request. The broker might have been temporarily stopped, for example.
- If the version of the connected Configuration Manager is earlier than Version 6.0. In this case, canceling deployment to a specific broker has no effect; you must cancel the entire domain deployment instead.

Deploying a message flow application

Before you start:

Before you can deploy a message flow application, you must have created, and started, a Configuration Manager. The broker added to the domain is actually a reference, so you must also create and start the physical broker on the target system. A WebSphere MQ listener must be running and you must also have created a domain, added a broker to that domain, and created an execution group within the broker. See the links to related tasks below for help with these.

Message flow applications are deployed to execution groups by adding required resources, optionally with their source files, to a broker archive (bar) file. The bar file is then sent to the appropriate Configuration Manager where it is unpacked and the individual files deployed to execution groups on individual brokers. The Message flow application deployment topic gives more details.

The tasks in this section describe the process:

1. Creating a server project
2. Creating a broker archive file
3. Adding files to a broker archive
 - Editing a broker archive file manually
 - Editing configurable properties
4. Deploying a broker archive file
5. Configuring a message flow at deployment time using UDPs
6. Checking the results of deployment

“Object version and keyword” on page 425 describes how to view version and keyword information about deployed objects.

The .lil file for a plugin node must also be deployed to every broker computer that uses that node in a message flow. For more details, see the section about developing user-defined extensions.

Creating a server project

Before you can deploy a message flow application, you must create a server project for it.

Before you start:

Save your message flow and message set projects.

Follow these steps to create a server project using the Message Brokers Toolkit.

1. Switch to the Broker Administration perspective.
2. Click **File** → **New** → **Other**.
3. Select **Show all wizards**, then in the list of wizards, expand **Server** and click **Server Project**.
4. Click **Next**.
5. If asked, click **OK** to enable “Base J2EE Support”.
6. Enter the name of your new server project.
7. Click **Finish**.

The folder that is created appears twice in the Navigator view (if **Show empty projects in Navigators** has been selected in the **Broker Administration Preferences** page):

- in the Domain Connections folder
- in the Broker Archives folder

Next:

Continue by creating a broker archive (bar) and adding files to it.

Creating a broker archive

Create a separate broker archive (bar) file for each configuration that you want to deploy.

There are two ways of creating a bar file:

- Using the Message Brokers Toolkit
- Using the `mqsicreatebar` command

Using the Message Brokers Toolkit

Before you start:

Either create a server project, or ensure that one already exists.

Follow these steps to create a bar file using the workbench:

1. From the Broker Administration perspective, click **File** → **New** → **Message Broker Archive**.
2. Enter the name of your server project, or select one from the displayed list.
3. Enter a name for the bar file that you are creating.
4. Click **Finish**.

A file with a `.bar` extension is created and is displayed in the Broker Administration Navigator view, under the Broker Archives folder. The Content editor for the bar file opens.

Next:

Continue by adding files to your broker archive and then deploying it.

Using the `mqsicreatebar` command

Follow these steps to create a bar file using the `mqsicreatebar` command:

1. Open a command window that is configured for your environment.
2. Enter the command, typed on a single line, using this as an example:

```
mqsicreatebar -b barName -p projectNames -o filePath
```

A file with a `.bar` extension is created.

The `-b` (bar file name), and `-o` (path for included files) parameters must be specified. The `-p` (project names) parameter is optional. The `mqsicreatebar` topic gives more details.

Next:

Continue by adding files to your broker archive and then deploying it.

Adding files to a broker archive

To deploy files to an execution group, you first include them in a broker archive (bar). The bar file is deployed by sending it to the Configuration Manager and from there, its contents are sent to the execution group on a broker.

You can only add message flows and message sets at the project level. However, after you have added the project to the bar file, you can use the **Remove** icon to remove individual message flows or message definitions, if required. Likewise, if you check the **Include message flow/set source** box, the source for all the message flows or message sets in the project are included but you can manually remove the source files for the message flows or message sets that you do not want.

To deploy XML, XSL, and JAR files inside a broker archive, the connected Configuration Manager and target broker must be Version 6.0 or later.

If there is a parent flow and subflow displayed in the **Add** dialogue, subflows are added automatically, so you only have to add the parent flow.

You can manually add XML, XSL, and JAR files by following these steps. However, JAR files that are required by JavaCompute nodes within message flows are added automatically from your Java project when you add the message flow. Similarly, XML and XSL files are automatically added if required by the flow.

It is not possible to read deployed files back from broker execution groups. Therefore, keep a copy of the deployed bar file, or of the individual files within it.

Follow these steps to add files to a broker archive (bar) file using the Message Brokers Toolkit:

1. Switch to the Broker Administration perspective.
2. Double-click your bar file in the Broker Administration Navigator view to open it. The contents of the bar file are shown in the Content editor. (If the bar file is new, this view is empty.)
3. Click the **Add** icon.
4. Check the boxes for the message flows, message sets, and other files that you want to include. (Duplicates within a bar file are automatically removed.)
5. Optional: If you want to include your message flow and message set source files, check the **Include message flow/set source** box.
6. Optional: If you want to compile ESQL so that it is compatible with Version 2.1 brokers, check the **Compile ESQL for brokers version 2.1** box.
7. Click **OK**.

A list of the files that are now in your bar file is displayed in the Content editor. You can choose not to display your message flow and message set source files by clearing the **Show source files** box at the bottom of the Content editor pane.

Next:

The next step is to deploy your broker archive (bar) file, but you might first want to edit configurable properties. You can also edit the contents of your bar file manually.

Editing a broker archive file manually

Before you start:

This task explains how to manually edit a broker archive (bar) file that already exists. If you have not already created a bar file, create it now, before continuing.

Follow these steps to edit a bar file manually using the Message Brokers Toolkit:

1. Export the bar file.
 - a. From the workbench, click **File** → **Export**. The Export window appears.
 - b. Select the export destination, such as a .zip file, and click **Next**.
 - c. Select the resources that you want to export and click **Next**.
 - d. Complete the destination information and click **Finish**. The file appears at the destination you specified as a .zip file.
2. Unzip the bar file.
3. Edit the properties that you want to change in an editor of your choice.
4. Save the file.
5. Import the bar file back into the workbench to deploy it.
 - a. From the workbench, click **File** → **Import**. The Import window appears.
 - b. Select **Zip file** from the list.
 - c. Click **Next**.
 - d. Specify the name and location of your bar file.
 - e. Select the server project that you want to contain the bar file.
 - f. Click **Finish**.

Next:

Continue by deploying your broker archive (bar) file.

Editing configurable properties

Before you start:

This task explains how to edit the configurable properties of your broker archive (bar) file deployment descriptor. If you have not already created a bar file, create it now, before continuing.

Follow these steps to edit properties using the Message Brokers Toolkit:

1. Switch to the Broker Administration perspective.
2. Select the **Configure** tab at the bottom of the Content editor pane. The properties that you can configure are listed.
3. Click the property for which you want to edit the value. The values that can be edited are displayed.
4. Replace the current value with the new value.
5. Save your bar file.

You can also edit this XML-format file manually using an external text editor or shell script.

Next:

Continue by deploying your broker archive (bar) file.

Deploying a broker archive file

Before you start:

This task explains how to deploy your broker archive (bar). If you have not already created a bar file, create it now, before continuing.

There are three ways of deploying a broker archive (bar) file:

- Using the Message Brokers Toolkit
- Using the `mqsideploy` command
- Using the Configuration Manager Proxy API

You need to have access rights if the execution group to which you want to deploy is restricted by an ACL.

Using the Message Brokers Toolkit

Follow these steps to deploy a bar file using the workbench:

1. Switch to the Broker Administration perspective.
2. Optional. Normally, an incremental bar file deployment is performed. If you want to perform a complete bar file deployment: right-click the target execution group in the Domains view and select **Remove Deployed Children**. Wait for the operation to complete before continuing.

It is not necessary to remove deployed children if you only want to refresh one or more of them with the contents of the bar file. The difference between a complete and an incremental bar file deployment is explained in the Message flow application deployment topic.

3. Click the bar file shown in the Navigator view to highlight it.
4. Drag the file onto your target execution group shown in the Domains view. Alternatively, right-click the bar file and click **Deploy file**. A dialog box shows all the domains, as well as execution groups within those domains to which the workbench is connected. A dialog box shows the execution groups (within their domains) to which you can deploy the bar file. Select an execution group and click **OK** to deploy the bar file. (Note: If you select a broker topology that is not connected to a domain, an attempt is made to connect it. If you click **Cancel**, the broker topology remains unconnected to a domain.)

Whichever method you use, you cannot select (and deploy to) more than one execution group at a time.

5. If the bar file has not been saved since it was last edited, you are asked whether you want to save it before deploying. If you click **Cancel**, the bar file is not saved and deployment does not take place.

The bar file is transferred to the Configuration Manager from where its contents (message flows and message sets, for example) are deployed to the execution group. In the Domains view, the assigned message flows and message sets are added to the appropriate execution group.

Next:

Continue by checking the results of the deployment.

Using the mqsideploy command

Follow these steps to deploy a bar file using the mqsideploy command:

1. Open a command window that is configured for your environment.
2. Using these as examples, enter the appropriate command, typed on a single line:

On z/OS:

```
/f MQ01CMGR,dp b=broker e=exngp a=barfile
```

This performs an incremental deployment. Add the `m=yes` parameter to perform a complete bar file deployment.

On other platforms:

```
mqsideploy -i ipAddress -p port -q qmgr -b broker -e exngp -a barfile
```

This performs an incremental deployment. Add the `-m` parameter to perform a complete bar file deployment.

The `-i` (IP address), `-p` (port), and `-q` (queue manager) parameters represent the connection details of the queue manager workstation, and on the z/OS console, MQ01CMGR is the name of the Configuration Manager component.

The `-b` (broker name), `-e` (execution group name), and `-a` (bar file name) parameters (or z/OS equivalent) must also be specified.

Next:

Continue by checking the results of the deployment.

Using the Configuration Manager Proxy API

Use the `deploy` method of the `ExecutionGroupProxy` class. By default, the `deploy` method performs an incremental deployment. To perform a complete deployment, use a variant of the method that includes the boolean `isIncremental` parameter; setting this to `false` indicates a complete deployment. (Setting it to `true` indicates an incremental deployment.)

For example:

```
import com.ibm.broker.config.proxy.*;
import java.io.IOException;

public class DeployTopology {
    public static void main(String[] args) {
        ConfigManagerConnectionParameters cmcp = new MQConfigManagerConnectionParameters("localhost", 1414, "QM1");
        try {
            ConfigManagerProxy cmp = ConfigManagerProxy.getInstance(cmcp);
            TopologyProxy t = cmp.getTopology();
            BrokerProxy b = t.getBrokerByName("BROKER1");
            ExecutionGroupProxy e = b.getExecutionGroupByName("default");
            e.deploy("deploy.bar");
        }
        catch (ConfigManagerProxyException cmpe) {
            cmpe.printStackTrace();
        }
        catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

Next:

Continue by checking the results of the deployment.

Configuring a message flow at deployment time using UDPs

User-defined properties (UDPs) give you the opportunity to configure message flows at deployment time, without modifying program code.

A UDP is a user-defined constant that you can use in your ESQL or Java programs. You can give the UDP an initial value when you declare it in your program, or when you use the Message Flow editor to create or modify a message flow.

In ESQL, you can define UDPs at the module or schema level.

For an overview of user-defined properties, see “User-defined properties” on page 50.

After a UDP has been defined by the Message Flow editor, you can modify its value before you deploy:

1. From the workbench, switch to the Broker Administration perspective.
2. Double click your bar file in the Broker Administration Navigator view. The contents of the bar file are shown in the Content editor.
3. Select the Configure tab at the bottom of the Content editor pane. This shows the names of your message flows; these can be expanded to show the individual nodes that are contained in the flow.
4. Click on a message flow name. The UDPs that are defined in that message flow are displayed with their values.
5. If the value of the UDP is unsuitable for your current environment or task, change it to the value that you want. The value of the UDP is set at the flow level and is the same for all eligible nodes that are contained in the flow. If a subflow includes a UDP that has the same name as a UDP in the main flow, the value of the UDP in the subflow is not changed.

Now you are ready to deploy the message flow. See “Deploying a broker archive file” on page 434.

Deploying a broker configuration

The broker configuration deployment overview explains when you might want to deploy a broker configuration.

There are three ways to deploy a broker configuration:

- Using the Message Brokers Toolkit
- Using the mqsideploy command
- Using the Configuration Manager Proxy API

Using the Message Brokers Toolkit

If you modify any multicast or interbroker settings with the workbench, a broker configuration deployment is automatically initiated when the changes are applied.

Using the mqsideploy command

Follow these steps to deploy a broker configuration using the mqsideploy command:

1. Open a command window that is configured for your environment.
2. Using these as examples, enter the appropriate command, typed on a single line, specifying the broker to which you want to deploy:

On z/OS:

```
/f MQ01CMGR,dp b=broker
```

On other platforms:

```
mqsideploy -i ipAddress -p port -q qmgr -b broker
```

The -i (IP address), -p (port), and -q (queue manager) parameters represent the connection details of the queue manager workstation, and on the z/OS console, MQ01CMGR is the name of the Configuration Manager component.

By specifying the broker to which you want to deploy (b= or -b), without indicating a bar file (-a), the broker configuration is deployed rather than a message flow application.

Next:

Continue by checking the results of the deployment.

Using the Configuration Manager Proxy API

Use the deploy method of the BrokerProxy class.

For example:

```
import com.ibm.broker.config.proxy.*;

public class DeployBrokerConfig {
    public static void main(String[] args) {
        ConfigManagerConnectionParameters cmcp =
            new MQConfigManagerConnectionParameters
                ("localhost", 1414, "QM1");
        try {
            ConfigManagerProxy cmp = ConfigManagerProxy.getInstance(cmcp);
            TopologyProxy t = cmp.getTopology();
            BrokerProxy b = t.getBrokerByName("BROKER1");
            if (b != null) {
                b.deploy();
            }
        }
        catch (ConfigManagerProxyException e) {
            e.printStackTrace();
        }
    }
}
```

Next:

Continue by checking the results of the deployment.

Deploying a publish/subscribe topology

Before you start:

Make sure that you have configured your broker domain.

The publish/subscribe topology deployment overview explains when you might want to deploy a topology and the difference between a complete and delta deployment.

There are three ways to deploy topology information:

- Using the Message Brokers Toolkit
- Using the `mqsidedeploy` command
- Using the Configuration Manager Proxy API

You can configure the workbench preferences so that topology information is automatically deployed after a change.

After you have deployed a publish/subscribe topology, you might see an extra execution group process called `$SYS_mqsi` in a process listing, or in the output from the `mqsilist` command. When you deploy a publish/subscribe topology for the first time, a new execution group process is started on your broker to handle the publish/subscribe messages. This execution group is only used internally: it does not appear in the workbench and you cannot deploy message flows to it. After you have deployed one or more of your own flows to another execution group, `$SYS_mqsi` is removed when the broker is subsequently restarted.

Using the Message Brokers Toolkit

Follow these steps to deploy a topology configuration using the workbench:

1. Switch to the Broker Administration perspective.
2. In the Domains view, expand the Domains from where you want to perform the deploy.
3. Right-click **Broker Topology** hierarchy.
4. Click **Deploy Topology Configuration**.
5. Click **Delta** to deploy only the changed items, or click **Complete** to deploy the entire configuration.

Alternatively, you can make a change to the Topology document in the Broker Administration perspective, save the changes and then select **Delta**. This behavior can be modified in the workbench preferences dialog.

The topology is deployed, and the Configuration Manager distributes it to the brokers in the domain.

Next:

Continue by checking the results of the deployment.

Using the `mqsidedeploy` command

Follow these steps to deploy a topology configuration using the `mqsidedeploy` command:

1. Open a command window that is configured for your environment.
2. Using these as examples, enter the appropriate command, typed on a single line:

On z/OS:

```
/f MQ01CMGR,dp l=yes
```

This performs a delta deployment. Add the `m=yes` parameter to deploy the entire configuration.

On other platforms:

```
mqsidedeploy -i ipAddress -p port -q qmgr -l
```

This performs a delta deployment. Add the `-m` parameter to deploy the entire configuration. The `-i` (IP address), `-p` (port), and `-q` (queue manager) parameters represent the connection details of the queue manager workstation, and on the z/OS console, MQ01CMGR is the name of the Configuration Manager component.

Next:

Continue by checking the results of the deployment.

Using the Configuration Manager Proxy API

Use the `deploy` method of the `TopologyProxy` class. By default, the `deploy` method performs a delta deployment. To deploy the complete hierarchy, use a variant of the method that includes the boolean `isDelta` parameter; setting this to `false` indicates a complete deployment. (Setting it to `true` indicates a delta deployment.)

For example:

```
import com.ibm.broker.config.proxy.*;

public class DeployTopology {
    public static void main(String[] args) {
        ConfigManagerConnectionParameters cmcp =
            new MQConfigManagerConnectionParameters
                ("localhost", 1414, "QM1");
        try {
            ConfigManagerProxy cmp =
                ConfigManagerProxy.getInstance(cmcp);
            TopologyProxy t = cmp.getTopology();
            t.deploy(false);
        }
        catch (ConfigManagerProxyException e) {
            e.printStackTrace();
        }
    }
}
```

Next:

Continue by checking the results of the deployment.

Deploying a publish/subscribe topics hierarchy

Before you start:

Make sure that you have configured your broker domain.

The topic deployment overview explains when you might want to deploy a topic hierarchy and the difference between a complete and delta deployment.

There are three ways to deploy a topics hierarchy:

- Using the Message Brokers Toolkit
- Using the `mqsidedeploy` command
- Using the Configuration Manager Proxy API

You can configure the workbench preferences so that a topics hierarchy is automatically deployed after a change.

Using the Message Brokers Toolkit

Follow these steps to deploy a topics hierarchy using the workbench:

1. Switch to the Broker Administration perspective.
2. In the Domains view, expand the Domains from where you want to perform the deploy.
3. Right-click **Topics** hierarchy.
4. Click **Deploy Topics Configuration**.
5. Click **Delta** to deploy only the changed items, or click **Complete** to deploy the entire configuration.

The topics hierarchy is deployed, and the Configuration Manager distributes the topics to brokers in the domain.

Next:

Continue by checking the results of the deployment.

Using the `mqsidedeploy` command

Follow these steps to deploy a topics hierarchy using the `mqsidedeploy` command:

1. Open a command window that is configured for your environment.
2. Using these as examples, enter the appropriate command, typed on a single line:

On z/OS:

```
/f MQ01CMGR,dp t=yes
```

This performs a delta deployment. Add the `m=yes` parameter to deploy the entire configuration.

On other platforms:

```
mqsidedeploy -i ipAddress -p port -q qmgr -t
```

This performs a delta deployment. Add the `-m` parameter to deploy the entire configuration. The `-i` (IP address), `-p` (port), and `-q` (queue manager) parameters represent the connection details of the queue manager workstation.

Next:

Continue by checking the results of the deployment.

Using the Configuration Manager Proxy API

Use the deploy method of the TopicRootProxy class. By default, the deploy method performs a delta deployment. To deploy the complete hierarchy, use a variant of the method that includes the boolean isDelta parameter; setting this to false indicates a complete deployment. (Setting it to true indicates a delta deployment.)

For example:

```
import com.ibm.broker.config.proxy.*;

public class DeployTopics {
    public static void main(String[] args) {
        ConfigManagerConnectionParameters cmcp =
            new MQConfigManagerConnectionParameters
                ("localhost", 1414, "QM1");
        try {
            ConfigManagerProxy cmp =
                ConfigManagerProxy.getInstance(cmcp);
            TopicRootProxy t = cmp.getTopicRoot();
            t.deploy(false);
        }
        catch (ConfigManagerProxyException e) {
            e.printStackTrace();
        }
    }
}
```

Next:

Continue by checking the results of the deployment.

Checking the results of deployment

After you have made a deployment, check that the operation has completed successfully. There are three ways of checking the results of a deployment:

- Using the Message Brokers Toolkit
- Using the mqsideploy command
- Using the Configuration Manager Proxy API

Also, check the system log on the target system where the broker was deployed to make sure that the broker has not reported any errors.

Using the Message Brokers Toolkit

Follow these steps to check a deployment using the workbench:

1. Switch to the Broker Administration perspective.
2. Expand the Domains view.
3. Double-click the **Event Log**.

When the deployment is initiated, an information message is displayed, confirming that the request was received by the Configuration Manager:

- BIP0892I

If the deployment completes successfully, you might also see one or more of these additional messages:

- BIP4040I

- BIP4045I
- BIP2056I

Using the mqsideploy command

The command returns numerical values from the Configuration Manager and any brokers affected by the deployment to indicate the outcome of the deployment. If it completes successfully, it returns 0. Refer to the mqsideploy topic for details of other values that you might see.

Using the Configuration Manager Proxy API

If you are using a Configuration Manager Proxy application, you can find out the result of a publish/subscribe topology deployment operation, for example, by using code similar to this:

```
TopologyProxy t = cmp.getTopology();

boolean isDelta = true;
long timeToWaitMs = 10000;
DeployResult dr = topology.deploy(isDelta, timeToWaitMs);

System.out.println("Overall result = "+dr.getCompletionCode());

// Display overall log messages
Enumeration logEntries = dr.getLogEntries();
while (logEntries.hasMoreElements()) {
    LogEntry le = (LogEntry)logEntries.nextElement();
    System.out.println("General message: " + le.getDetail());
}

// Display broker specific information
Enumeration e = dr.getDeployedBrokers();
while (e.hasMoreElements()) {

    // Discover the broker
    BrokerProxy b = (BrokerProxy)e.nextElement();

    // Completion code for broker
    System.out.println("Result for broker "+b+" = " +
        dr.getCompletionCodeForBroker(b));

    // Log entries for broker
    Enumeration e2 = dr.getLogEntriesForBroker(b);
    while (e2.hasMoreElements()) {
        LogEntry le = (LogEntry)e2.nextElement();
        System.out.println("Log message for broker " + b +
            le.getDetail());
    }
}
}
```

The `deploy()` method blocks until all affected brokers have responded to the deployment request.

When the method returns, the `DeployResult` represents the outcome of the deployment at the time when the method returned; the object is not updated by the Configuration Manager Proxy.

If the deployment message could not be sent to the Configuration Manager, a `ConfigManagerProxyLoggedException` is thrown at the time of deployment. If the Configuration Manager receives the deployment message, then log messages for

the overall deployment are displayed, followed by completion codes specific to each broker affected by the deployment. The completion code is one of the following static instances from the `com.ibm.broker.config.proxy.CompletionCodeType` class:

Completion code	Description
pending	The deploy is held in a batch and will not be sent until you issue <code>ConfigManagerProxy.sendUpdates()</code> .
submitted	The deploy message was sent to the Configuration Manager but no response was received before the timeout occurred.
initiated	The Configuration Manager replied stating that deployment has started, but no broker responses were received before the timeout occurred.
successSoFar	The Configuration Manager issued the deployment request and some, but not all, brokers responded with a success message before the timeout period expired. No brokers responded negatively.
success	The Configuration Manager issued the deployment request and all relevant brokers responded successfully before the timeout period expired.
failure	The Configuration Manager issued the deployment request and at least one broker responded negatively. You can use <code>getLogEntriesForBroker</code> for more information on why the deployment failed.
notRequired	A deployment request was submitted to the Configuration Manager involved with the supplied broker, but the request was not sent to the broker because its configuration is already up to date.

Canceling a deployment that is in progress

Before you start:

Canceling a deployment should only be a last resort if you are sure that a broker, or several brokers in a domain, will never be able to process a previous deployment request. For this reason, make sure that you understand the implications of this action, described in the [Cancel deployment overview](#) topic.

It is possible to cancel all outstanding deployments in the domain, or just those to a particular broker.

- When canceling deployment across the domain, you must have full access on the Configuration Manager.
- When canceling deployment to a specific broker, you must have full access on that broker.

If you want to ensure that previous deployment messages are not processed when an affected broker is restarted, first remove any deployment messages:

1. Stop the broker
2. Check the broker's `SYSTEM.BROKER.ADMIN.QUEUE` and `SYSTEM.BROKER.EXECUTIONGROUP.QUEUE`, and manually remove any deployment messages.
3. Proceed to cancel the deployment.

There are three ways to cancel a deployment:

- Using the Message Brokers Toolkit

- Using the `mqsidedeploy` command
- Using the Configuration Manager Proxy API

Using the Message Brokers Toolkit

Follow these steps to cancel the deployment to a particular broker or all outstanding deployments in a domain, using the workbench:

1. Switch to the Broker Administration perspective.
2. In the Domains view, right-click either a particular broker or a connected domain.
3. Click **Cancel Deployment**.

Deployments to the broker or domain are canceled.

Next:

Continue by checking the results. (A BIP0892I information message is displayed to show that the request was received by the Configuration Manager.)

Using the `mqsidedeploy` command

Follow these steps to cancel a deployment using the `mqsidedeploy` command:

1. Open a command window that is configured for your environment.
2. Using these as examples, enter the appropriate command, typed on a single line:

On z/OS:

```
/f MQ01CMGR,dp t=yes b=B1
```

This cancels deployment to the broker called B1. Omit the `b` argument to cancel all outstanding deployments in the domain.

On other platforms:

```
mqsidedeploy -i ipAddress -p port -q qmgr -c -b B1
```

This cancels deployment to the broker called B1. Omit the `-b` parameter to cancel all outstanding deployments in the domain. The `-i` (IP address), `-p` (port), and `-q` (queue manager) parameters represent the connection details of the queue manager workstation, and on the z/OS console, MQ01CMGR is the name of the Configuration Manager component.

Next:

Continue by checking the results. (A BIP0892I information message is displayed to show that the request was received by the Configuration Manager.)

Using the Configuration Manager Proxy API

To cancel all outstanding deployments in a domain, use the `cancelDeployment` method of the `ConfigManagerProxy` class. For example:

```
public class CancelAllDeploys {
    public static void main(String[] args) {
        ConfigManagerConnectionParameters cmcp =
            new MQConfigManagerConnectionParameters
                ("localhost", 1414, "QM1");
        try {
            ConfigManagerProxy cmp =
```

```

        ConfigManagerProxy.getInstance(cmcp);
        cmp.cancelDeployment();
    }
    catch (ConfigManagerProxyException e) {
        e.printStackTrace();
    }
}
}

```

To cancel deployment to a specific broker in a domain, use the `cancelDeployment` method of the `BrokerProxy` class. For example, to cancel deployment to a broker called *B1*:

```

import com.ibm.broker.config.proxy.*;

public class CancelDeploy {
    public static void main(String[] args) {
        ConfigManagerConnectionParameters cmcp =
            new MQConfigManagerConnectionParameters
                ("localhost", 1414, "QM1");
        try {
            ConfigManagerProxy cmp =
                ConfigManagerProxy.getInstance(cmcp);
            TopologyProxy t = cmp.getTopology();
            BrokerProxy b = t.getBrokerByName("B1");
            b.cancelDeployment();
        }
        catch (ConfigManagerProxyException e) {
            e.printStackTrace();
        }
    }
}

```

Next:

Continue by checking the results. (A BIP0892I information message is displayed to show that the request was received by the Configuration Manager.)

Renaming objects that are deployed to execution groups

You cannot rename an object while it is still deployed to an execution group. Instead, first remove the deployed object from the execution group. Then, having renamed it, deploy it again.

Removing a deployed object from an execution group

There are three ways of removing deployed objects from an execution group:

- Using the Message Brokers Toolkit
- Using the `mqsidedeploy` command
- Using the Configuration Manager Proxy API

Using the Message Brokers Toolkit

Follow these steps to remove an object from an execution group using the workbench:

1. Switch to the Broker Administration perspective.
2. From the Domains view, right-click the object that you want to remove.
3. Click **Remove** from the pop-up menu, and **OK** to confirm.

An automatic deployment is performed for the updated broker and a BIP08921 information message is produced, confirming that the request was received by the Configuration Manager.

Using the mqsideploy command

Follow these steps to remove an object from an execution group using the mqsideploy command:

1. Open a command window that is configured for your environment.
2. Using these as examples, enter the appropriate command, typed on a single line:

On z/OS:

```
/f MQ01CMGR,dp t=yes b=broker e=execgp d=file1.cmf:file2.dictionary:file3.xml
```

On other platforms:

```
mqsideploy -i ipAddress -p port -q qmgr -b broker -e execgp  
-d file1.cmf:file2.dictionary:file3.xml
```

Optionally, specify the -m option to clear the contents of the execution group. This tells the execution group to completely clear any existing data before the new bar file is deployed. The -i (IP address), -p (port), and -q (queue manager) parameters represent the connection details of the queue manager workstation, and on the z/OS console, MQ01CMGR is the name of the Configuration Manager component.

The -d argument (or d= argument on z/OS) is a colon separated list of files to be removed from the named execution group. Invoking the command above causes the deployed objects (file1.cmf, file2.dictionary and file3.xml) to be removed from the specified execution group and broker.

The command displays feedback as responses are received from the Configuration Manager and any brokers affected by the deployment. If the command completes successfully, it returns 0.

Using the Configuration Manager Proxy API

One way of removing deployed objects using the Configuration Manager Proxy API is to get a handle to the relevant ExecutionGroupProxy object and then invoke its deleteDeployedObjectsByName() method. For example:

```
import com.ibm.broker.config.proxy.*;  
  
public class DeleteDeployedObjects {  
    public static void main(String[] args) {  
        ConfigManagerConnectionParameters cmcp =  
            new MQConfigManagerConnectionParameters  
                ("localhost", 1414, "QM1");  
        try {  
            ConfigManagerProxy cmp =  
                ConfigManagerProxy.getInstance(cmcp);  
            TopologyProxy t = cmp.getTopology();  
            BrokerProxy b = t.getBrokerByName("broker1");  
            ExecutionGroupProxy e =  
                b.getExecutionGroupByName("default");  
            e.deleteDeployedObjectsByName(  
                new String[] { "file1.cmf",  
                    "file2.dictionary",  
                    "file3.xml" }, 0);  
        }  
    }  
}
```

```
        catch (ConfigManagerProxyException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Part 3. Debugging

Testing and debugging message flow applications	451
Debugging a message flow	451
Flow debugger	452
Starting the debugger	455
Working with breakpoints	462
Stepping through flow instances	466
Debugging data	470
Managing flows and flow instances	473

Testing and debugging message flow applications

Use the flow debugger to test and error-check your message flows.

This section includes the following topics:

- “Debugging a message flow”

Debugging a message flow

This topic lists all the tasks involved in debugging a message flow. If you are new to debugging, see “Flow debugger overview” on page 452 for a conceptual overview.

1. Starting the flow debugger:
 - a. “Setting flow debug preferences” on page 456
 - b. “Attaching to the flow engine” on page 456
 - c. “Putting a test message” on page 459
 - d. “Getting a test message” on page 461
2. Working with breakpoints:
 - a. “Adding breakpoints” on page 462
 - b. “Restricting breakpoints to specific flow instances” on page 463
 - c. “Disabling breakpoints” on page 464
 - d. “Enabling breakpoints” on page 464
 - e. “Removing breakpoints” on page 465
3. Stepping through flow instances:
 - a. “Resuming flow instance execution” on page 466
 - b. “Running to completion” on page 466
 - c. “Stepping over nodes” on page 467
 - d. “Stepping into subflows” on page 467
 - e. “Stepping out of subflows” on page 467
 - f. “Stepping over source code” on page 468
 - g. “Stepping into source code” on page 469
 - h. “Stepping out of source code” on page 469
4. Debugging data:
 - a. “Debugging messages” on page 470
 - b. “Debugging ESQL” on page 471
 - c. “Debugging Java” on page 472
 - d. “Debugging mappings” on page 472
5. Managing flows and flow instances:
 - a. “Querying a broker for deployed flows” on page 473
 - b. “Redeploying a flow” on page 473
 - c. “Terminating a flow instance” on page 474
 - d. “Detaching from the flow engine” on page 474

Flow debugger

Use the flow debugger to test your message flows. The visual interface makes it straightforward to catch the flow at any point and examine or alter the message state before stepping to the next point.

This section includes the following topics:

- “Flow debugger overview”
- “ESQL nodes” on page 454
- “Java nodes” on page 455
- “Mapping nodes” on page 455

Flow debugger overview

This topic introduces the flow debugger and contains the following sections:

- “Introduction”
- “The Debug perspective” on page 453
- “The Debug view” on page 453
- “The Message Flow editor” on page 454
- “The Breakpoints view” on page 454
- “Variables view” on page 454

After reading this conceptual overview, find detailed instructions to get you going in the task “Debugging a message flow” on page 451.

Introduction: The flow debugger is the tool that you use in the workbench to visually debug message flows. Read this topic to gain an understanding of the flow debugger before starting to use it.

Also, to use the flow debugger effectively, you need to have a basic understanding of message flows and their representation in the workbench, see “Message flows overview” on page 3 for an introduction.

What you can do: The flow debugger features a visual debugging environment. You can set breakpoints in a flow and then step through the flow. While you are stepping, you can examine and change the message and variables in ESQL code or Java code. These capabilities enable you to debug a wide variety of error conditions in flows, such as:

- Incorrectly wired activities (for example, outputs that are connected to the wrong inputs).
- Incorrect conditional branching in transition conditions.
- Unintended infinite loops in flow.

From a single workbench, the debugger can attach to more than one execution group, and debug multiple flows in different execution groups (and therefore multiple messages) at the same time. However, a deployed flow in one execution group can be debugged by only one user at a time. If you attach your debugger to a flow, another user can attach to the same Data Flow Engine, but they will not see the flow that you are debugging.

How to start debugging:

- Deploy your message flow to an execution group in a broker and make sure that the broker is running.

- Make sure that the computer hosting the broker has IBM Agent Controller installed and running (you can debug a flow in a broker hosted on this computer or a remote one). The IBM Agent Controller (RAC) authorizes user access to debug flows.
- Attach the flow debugger to the flow engine (execution group).

The Debug perspective: Use the Debug perspective for debugging. In its default mode, this perspective shows the following four windows:

- Debug view
- Breakpoints view (shared pane)
- Variables view (shared pane)
- Message Flow editor

First use the **Debug (Create, manage, and run configurations)** wizard to attach the flow debugger to the flow engine (execution group) where your flow is deployed. (See “Attaching to the flow engine” on page 456 for more details on how to start and complete the wizard).

Then use the various views in the Debug perspective to debug, as summarized in the following sections.

The Debug view: When you attach the flow debugger to the flow runtime engine, the Debug view displays the names of the following flow-related entities:

- The host computer and the flow runtime engine that it is running. This is shown as a concatenation of the names of the following, delimited by colons.
 - The host computer
 - The broker
 - The execution group
 - The flow engine

The entry is identified by the flow runtime engine symbol. For example:

–  TestPC01:WMQIV5BR:TestExecution:DataFlowEngine


- The flows that are deployed in the flow runtime engine, identified by the flow symbol, for example:

–  TestFlow

- The flow instances that have been created for each flow, each identified by one of the following symbols, for example:

–  3068 (Paused).

–  3068 (Running).

–  3068 (Terminated).

In the Debug view, you can perform the following debugging tasks:

- Query a flow runtime engine for currently deployed flows
- Detach a flow runtime engine from the flow debugger
- Resume flow execution
- Run to termination
- Step over a node
- Step into or out of a subflow




- Step over, into, or out of, source code

The Message Flow editor: The Message Flow editor has different functions in different perspectives:

- In the Broker Application Development perspective, the Message Flow editor is used to create, graphically display, and edit flows.
- In the Debug perspective, the Message Flow editor is used only to graphically display and debug flows, as described in this topic.

For details of the other uses of this editor, see the description in Message Flow editor and the tasks in “Defining message flow content” on page 135.

In addition to displaying a flow, the Message Flow editor also displays any breakpoints that are set in the flow. Each breakpoint is identified by a symbol as follows:

-  breakpoint enabled.
-  breakpoint disabled.
-  flow paused at breakpoint.



Also the editor displays the following symbol above a node:

-  flow paused at a node containing ESQL code or Java code that the flow debugger can step into.

In the Message Flow editor, you can add or remove breakpoints.

The Breakpoints view: The Breakpoints view and the Variables view share the same pane. Click one of the tabs to select the view that you want.

The Breakpoints view displays the breakpoints that are set in all instances of a selected flow. Each breakpoint is identified by one of two symbols (as also used in the Message Flow editor) as follows:

-  breakpoint enabled.
-  breakpoint disabled.

In the Breakpoints view, you can perform the following debugging tasks:

- Remove breakpoints
- Disable or enable breakpoints
- Restrict breakpoints to one or more instances of a flow

Variables view: The Variables view and the Breakpoints view share the same pane. Click one of the tabs to select the view that you want.

The Variables view displays the messages that are currently traveling through the flow. Use the view to examine or change the content of a message in a flow during debugging.

ESQL nodes

In a message flow, the types of node listed below can contain ESQL code:

- Compute node

- Filter node
- Database node

When you want to debug the ESQL code, set a breakpoint just before the node. Then, in the flow debugger, when the flow pauses at the breakpoint, you can step into the code, and then step through it (step over) line by line. You can also examine and change the ESQL variables.

For more information on how to do this, see the Related tasks links below.

Java nodes

In a message flow, the types of node listed below can contain Java code:

- JavaCompute node
- User-defined node

When you want to debug the Java code, set a breakpoint just before the node. Then, in the flow debugger, when the flow pauses at the breakpoint, you can step into the code, and then step through it (step over) line by line. You can also examine and change the Java variables.

For more information on how to achieve this, see the related tasks links below.

Mapping nodes

You access and maintain mappings through the types of node listed below:

- Mapping node
- DataInsert node
- DataUpdate node
- DataDelete node
- Extract node
- Warehouse node

When you are debugging mapping nodes, the flow debugger allows you to step into mappings and step over mappings. You can set breakpoints on any of the mapping commands.

When you want to debug a flow with mappings, set a breakpoint at the connection right before the mapping node. Then, in the flow debugger, when the flow pauses at the breakpoint, you can step into the code, and then step through it (step over) line by line. You can view mapping variables, and you can view and alter your own user-defined variables.

For more information on how to do this, see the Related tasks links below.

Starting the debugger

This section describes the tasks that are involved in starting the flow debugger. The section includes the following topics:

- “Setting flow debug preferences” on page 456
- “Attaching to the flow engine” on page 456
- “Putting a test message” on page 459
- “Getting a test message” on page 461

Setting flow debug preferences

The following steps show you how to set your own preferences for the flow debugging environment.

1. Click **Window** → **Preferences** to invoke the Preferences wizard.
2. In the left frame of the wizard, select **Message Broker Debug** to open the Flow Debugger preferences page.
3. Make your selections.
4. Click **OK**.

Attaching to the flow engine

Before you start

To complete this task, you must have completed the following tasks:


- Install IBM Rational Agent Controller on the computer where the broker is running; see *Installing Rational Agent Controller*
- Create a message flow; see “Developing message flows” on page 3
- Deploy the flow; see “Deploying” on page 419
- Ensure that the broker where your flow is deployed is running.

To debug a deployed flow:


Before you can debug your message flow, you must attach the flow debugger to the flow engine (execution group) where your flow is deployed. If you want to, you can attach the flow debugger to multiple flow engines that are running on the same or different host machines and then simultaneously debug their flows.

To attach to the flow engine:

1. Switch to the Broker Administration perspective. Note the name of your message flow as displayed in the **Domains** pane.
2. Open that flow into the Message Flow editor by double-clicking its name in the **Broker Administration Navigator** pane.
3. Add a breakpoint to a connection that leads out of the input node of the message flow. Adding this breakpoint ensures that the message flow does not run to completion before you can begin to debug it.

The breakpoint appears as . (For information about adding a breakpoint, see “Working with breakpoints” on page 462).

4. Switch to the Debug perspective.
5. Click the down-arrow on the **Debug** icon  on the toolbar, and select **Debug** to invoke the **Debug (Create, manage, and run configurations)** wizard.

Tip: You are creating a debug launch configuration. If you have already created one, you can relaunch it by clicking directly on the **Debug** icon  itself. Note, however, that this will generate an error if:

- You have not created a debug launch configuration
- The broker and execution group you previously attached to are no longer running
- The broker and execution group have been restarted and hence have a new *Process ID* (see below).

6. In the list of configurations, select **Message Broker Debug** and click the **New** button. A set of tabbed panels appears in the window, beginning with **Connect**.

Tip: The **Debug** button remains grayed out until you complete the fields on the **Connect** panel. After that you can choose to complete the fields on the other panels, or go straight to clicking **Debug**.

The panels in the wizard are as follows:-

- a. **Connect** - use this panel to establish a connection to the flow engine through the IBM Agent Controller. You must complete all the fields on this panel before you can click the **Debug** button to start a debug session

IBM Agent Controller port number

enter the port number you want to use.

Flow Project

select your flow project.

HostName

select the host computer that the flow engine is running on. If the host is not listed, enter the host name or IP address of the host computer in the **HostName** field (if the **HostName** field is not available, first click the **Reset** button, then make your changes).

Flow Engine

select the broker and flow engine you want to debug. In the list box that opens when you click the **Browse** button, each flow engine is listed as its process number, followed by the name of the broker and the name of the execution group separated by a colon, for example:

ProcessID	Engine name
-----	-----
5984	WMBV6BR:default

If the flow engine does not appear in the list box, click **Refresh** to update the list box with the names of all flow engines that are currently deployed and available on the host computer. (If the flow engine still does not appear in the list box, try restarting IBM Agent Controller on the host computer).

Note: You will be presented with the option of attaching to any execution group running on the target host. This includes execution groups that do not have any flows deployed.

Tip: The process number is the Windows *PID*, as listed in the **Task Manager** on the **Processes** page. The PID was reported to you in the Event Log when you deployed, as described in "Deploying a publish/subscribe topology" on page 438.

You can now click **Debug** to go the next step, or you can go on to complete the other panels as follows.

- b. **Java debug setting** - use this panel when you want to debug a Java plugin node or Java compute node. The Java port is the port number that is specified for the broker JVM. If you do not specify a port, Java debugging is disabled.


Tip:

Setting the broker JVM debug port

In order to debug a JavaCompute node, or a user defined node containing Java code, the broker JVM must be configured with a debug port number. To do this issue the following command (all on one line):

```
mqschangeproperties <broker-name> -e <execution-group-name>  
-o ComIbmJVMManger -n jvmDebugPort -v <port-number>
```

The broker must be restarted after this command has been issued.

- c. **Source** - use this panel to tell the debugger where to look for your source files for flow, mapping, ESQL, or Java during debugging. The lookup path can be an eclipse project name, external folder, or a zipped file. You can specify multiple locations, but the debugger will always first look in the message flow project that you specified on the **Connect** panel.
 - d. **Common** - this panel is not directly used by the flow debugger, however if you set options on it they will take effect. See the Workbench User Guide for details.
7. Click the **Debug** button. In the Debug view, the names of the selected host computer and flow engine are displayed.
 8. When the next message comes into your flow and arrives at the breakpoint, the flow pauses, the breakpoint icon changes to  , and you can start debugging.
 9. In the **Debug view**, double-click the message flow that you want to debug. The message flow opens in the Message Flow editor, and now you can add more breakpoints, start stepping over the flow, and so on.

Tip: From a single workbench, the debugger can attach to more than one execution group, and debug multiple flows in different execution groups (and therefore multiple messages) at the same time. However, a deployed flow in one execution group can be debugged by only one user at a time, so if you attach your debugger to it, another user cannot also attach a debugger at the same time.

Note: The flow debugger provided in Version 6.0 can debug runtime brokers from previous versions. You can select the version of the broker that you want to debug by checking the corresponding option on the **Engine Selection** panel in the **Debug** wizard as described above.

The following table shows what is supported on different platforms. **RAC** is the short name for **IBM Agent Controller**.

	Version 6.0 Broker + RAC 6.0	Version 5 Broker + Fix Pack 3 or later + RAC 5.0.2	Version 5 Broker + Fix Pack 2 + RAC 5.0.2	Version 5 Broker + Fix Pack 2 + RAC 5.0.1
Windows	Yes	Yes	Yes	Yes
AIX	Yes	Yes	Yes	Yes
Solaris	Yes	Yes	Yes	No
HP-UX	Yes	Yes	No	No
z/OS	Yes	Yes	No	No
Linux	Yes	Yes (with IFix)	No	No

Contact your IBM Support Center if you want to use the flow debugger with a Fix Pack 3 broker on Linux.

Putting a test message

Before you start

Complete the following tasks:

- Creating a message flow; see “Developing message flows” on page 3
- “Attaching to the flow engine” on page 456.

This topic explains how to use a test message to help you debug. It contains the following sections:

- “Introduction”
- “To configure and use an enqueue file”
- “Optional: To add data to your message” on page 460
- “Optional: To use a file of sample data” on page 460
- “Optional: To create a file of sample data for the message” on page 460
- “Optional: To import an existing file of sample data for the message” on page 461


Introduction:

To start debugging your message flow, you might want to trigger the flow by putting a test message onto the input queue for your first MQInput node. This topic tells you how to put a test message by configuring and using an enqueue file in the workbench. This is an easy and repeatable alternative to using the WebSphere MQ Explorer or command line interfaces to put a message.

Tip: This topic also describes various ways of adding test data to a message. If you want just a small amount of test data in your test message, type the data into a window. If you want your test message to contain a larger quantity of sample data (for example some structured XML), first create or import a file containing that data, then get the enqueue file to use it. Follow the optional sections and steps below to use either of these methods of creating and adding data.


To configure and use an enqueue file:

To configure an enqueue file so that you can use it to send a test message:

1. Switch to the Broker Administration perspective.
2. On the workbench toolbar, click the arrow on the **Put a message onto a queue** icon  .
3. On the drop-down menu, click **Put Message...** to invoke the New Enqueue Message File wizard.
4. Select the message flow project containing the message flow that you are debugging.
5. In the **File name** field, enter a name for the file to create (the extension `.enqueue` is added automatically).
6. Click **Finish**. The enqueue file is created, and a view opens showing its details.

7. Enter the names for the queue manager and the queue for the input node for this flow.

Note: Queue manager names are case-sensitive.

8. For Version 6.0, enqueue supports putting a message on a remote queue. Enter values to identify the host and port of the computer that is hosting the queue.
9. Click **File** → **Save** to save the enqueue file.
10. Optional: To put the message to the queue immediately from this window, click the **Write to queue** button.
11. Click the arrow on the **Put a message onto a queue** icon  to see your enqueue file listed on the drop-down menu.
12. Click this file on the menu, (or if it is number 1 on the menu, just click the icon itself) to put a message to the queue. If you have set appropriate breakpoints, the flow debugger pauses the flow at the next one.

To find your enqueue file at a later time, switch to the Broker Application Development perspective and expand the navigation tree for your message flow project. Double-click your enqueue file to open it in a view.

Optional: To add data to your message:

To quickly add some test data to a message:

1. Open your enqueue file and select the **Browse** tab.
2. Type your test data directly into the “Message data” window.
3. Put the test message by selecting the **General** tab and clicking the **Write to queue** button.

Optional: To use a file of sample data:

To get the enqueue file to use a file with sample data that you have created or imported as described in the two sections below:

1. Open your enqueue file; at **File name** click the **Browse** button.
2. In the “Add a message” window, select your file and click **OK**.
3. Click **File** → **Save** when you have finished.
4. To see the data in your file, select the **Browse** tab at the bottom of the enqueue view. If you want to change the data, either type some text into the “Message data” window, or edit the file by double-clicking it in the Resource Navigator view.

Tip: If you decide to experiment with using an XML data file instead of text, do not forget to edit the properties for the input node of your message flow and set the **Message Domain** to **XML**. When your message appears in the Flow Debugger in the Variables view, the XML is parsed and expandable in the tree.

Optional: To create a file of sample data for the message:

To create a new file of sample data:

1. Switch to the Broker Application Development perspective.
2. Click **File** → **New** → **Other...**
3. Click **Simple**, then **File**.

4. In the New File window, select the project in which you want to keep the file, then at **File name** enter a name and extension for your file. If your file contains XML, make sure that the extension is .XML.
5. Click **Finish**. The file is created and appears in the Resource Navigator view. A view opens with an appropriate editor (text or XML) for the file.
6. Edit the file and enter the text or XML data that you want in it.
7. Click **File** → **Save** when you have finished.

You can now select the file as described in the “Optional: To add data to your message” on page 460 section. You can also double-click the file to open it in an appropriate editor.

Optional: To import an existing file of sample data for the message:

If you already have a file on your computer containing sample data that you want to use in a test message, use these steps to import the data into the workbench. If the file contains XML, make sure that it has the extension .XML.

Note: After you have imported the file as described here, a copy of your file is stored in the workbench data space along with all other workbench files for your configuration. Your original file will not be used directly again.

To import an existing file of sample data:

1. Switch to the Broker Application Development perspective.
2. Click **File** → **Import**.
3. In the list of wizards, select **File system** then click **Next**.
4. At **Directory**, click the **Browse** button and navigate to the folder containing your file then click **OK**. A list of the files in the folder appears.
5. In the list of files, select the check box next to the file that you want to import.
6. At **Folder**, click the **Browse** button and select the project in which you want to keep the file.
7. Click **Finish**.

The file appears in the Resource Navigator in your chosen project, and you can select it as described in the “Optional: To add data to your message” on page 460 section on this page. You can also double-click it to open it in an appropriate editor.

Getting a test message



Before you start

Complete the following tasks:

- Creating a message flow; see “Developing message flows” on page 3
- “Attaching to the flow engine” on page 456.

While you are debugging your message flow, you might want to read a message from the queue on your MQOutput node. This topic tells you how to get a test message by configuring and using a dequeue file in the workbench. This is an easy and repeatable alternative to using the WebSphere MQ Explorer or command line interfaces to get a message.

To create and use a dequeue configuration:

1. Switch to the Debug perspective.
2. On the workbench toolbar, click the arrow on the **Get a message from a Queue** icon .
3. On the drop-down menu, click **Get Message...** to invoke the Dequeue Message window.
4. Enter the names for the queue manager and (output node) queue that you want.
5. Click **Read From Queue** to read a message from the queue.
6. After closing the window, the next time that you click the arrow on the **Get a message from a Queue** icon , you see the dequeue configuration that you created above listed on the drop-down menu. Click this file on the menu (or if it is number 1 on the menu, just click the icon itself) to get a message from the queue.
7. Optional: While the Dequeue Message window is open, you can save a read message into a file in your message flow project to keep it for later. Click the **Save As...** button, and in the Save Message As window, select the flow project and enter a name for the file, including a .extension if you want one.

Tip: If you saved any messages into files and want to see these saved messages, switch to the Broker Application Development perspective and expand the navigation tree for your flow project. Double-click a saved message file to open it in a view.

Working with breakpoints

This section describes working with breakpoints. It includes the following topics:

- “Adding breakpoints”
- “Restricting breakpoints to specific flow instances” on page 463
- “Disabling breakpoints” on page 464
- “Enabling breakpoints” on page 464
- “Removing breakpoints” on page 465

Adding breakpoints

Before you start

Completed the following task:

- “Attaching to the flow engine” on page 456

In the Debug perspective, you can add breakpoints to the connections of a message flow that is open in the Message Flow editor. Any breakpoint that is added to a flow is also automatically added to all other instances of the flow and you do not need to restart any of the instances.

When you add a breakpoint to a connection, the connection is flagged with the enabled breakpoint symbol .

To add breakpoints to the connections of a message flow:

1. Switch to the Debug perspective.

2. Add breakpoints to the appropriate connections. Use any of the following methods:

Option	Description
Add breakpoints individually to selected connections:	<ol style="list-style-type: none"> 1. In the Message Flow editor, right-click the connection where you want to set the breakpoint. 2. Click Add Breakpoint on the pop-up menu.
Add breakpoints simultaneously to all connections entering a selected node:	<ol style="list-style-type: none"> 1. In the Message Flow editor, right-click the node before which you want to set breakpoints. 2. Click Add Breakpoints Before Node on the pop-up menu.
Add breakpoints simultaneously to all connections leaving a selected node:	<ol style="list-style-type: none"> 1. In the Message Flow editor, right-click the node after which you want to set breakpoints. 2. Click Add Breakpoints After Node on the pop-up menu.

Now that you have added some breakpoints, you can step through the flow instance and work with data, as described in:

- “Stepping through flow instances” on page 466
- “Debugging data” on page 470

Restricting breakpoints to specific flow instances

Before you start

Complete the following task:

- “Adding breakpoints” on page 462

In the Debug perspective, when you add a breakpoint to a message flow in the Message Flow editor, the breakpoint automatically applies to all instances of the flow. However, you can choose to restrict a breakpoint to one or more instances of a flow. This enables you to work more easily with just those instances that you are currently interested in, rather than with all instances.

To restrict a breakpoint to one or more flow instances:

1. In the Breakpoints view, right-click the breakpoint that you want to restrict, then click **Properties** to open the Flow Breakpoints Properties window.
2. In the **Restrict to Selected Flow Instance(s)** list box, select the check boxes of those instances to which you want to restrict the breakpoint.

You must have at least one instance active, otherwise the **Restrict to Selected Flow Instance(s)** list box will contain no check boxes.

If any instance is currently paused on the breakpoint, all check boxes in the **Restrict to Selected Flow Instance(s)** list box will be grayed out and you will be unable to select them.

3. Click **OK**.

Now you can add additional breakpoints (if needed), step through the instance, and work with data.

Disabling breakpoints

Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462

In the Debug perspective, you can disable any breakpoints that are currently enabled. The following symbol identifies enabled breakpoints:



To disable breakpoints:

1. Switch to the Debug perspective.
2. In the Breakpoints view, select one or more enabled breakpoints that you want to disable.
3. Right-click the selected breakpoints, then click **Disable**. The breakpoints are disabled in all instances of the message flow where they are set.
4. Optional: You can also disable a single breakpoint by right-clicking it, then clicking **Properties** on the pop-up menu. Then clear the **Enabled** check box and click **OK**. The breakpoint is disabled in all instances of the message flow where it is set.

When you disable a breakpoint, the breakpoint symbol changes to the following symbol:



Enabling breakpoints

Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Disabling breakpoints”

In the Debug perspective, you can enable any breakpoints that are currently disabled. The following symbol identifies disabled breakpoints:



To enable breakpoints:

1. Switch to the Debug perspective.
2. In the Breakpoints view, select one or more disabled breakpoints that you want to enable.
3. Right-click the selected breakpoints then click **Enable**. The breakpoints are enabled in all instances of the message flow where they are set.
4. Optional: You can also enable a single breakpoint by right-clicking on it then clicking **Properties**. Then select the **Enabled** check box and click **OK**. The breakpoint is enabled in all instances of the message flow where it is set.

When you enable a breakpoint, the breakpoint symbol changes to the following symbol:



Removing breakpoints

Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462





In the Debug perspective you can remove breakpoints either where they are shown on the connections of a message flow that is open in the Message Flow editor, or where they are listed in the Breakpoints view. The following icon shows the breakpoint:



Any breakpoint that is removed from a message flow is also automatically removed from all other instances of the message flow where it is set.

To remove breakpoints:

1. Switch to the Debug perspective.
2. Remove the breakpoints. Use one of the following methods, depending on how many breakpoints you want to remove:

Option	Description
Remove individual breakpoints from connections in the Message Flow editor:	<ol style="list-style-type: none"> 1. In the Message Flow editor, right-click the breakpoint that you want to remove, then click Remove Breakpoint.
Remove selected breakpoints simultaneously in the Breakpoints view:	<ol style="list-style-type: none"> 1. Click the Flow Breakpoints tab to show the Breakpoints view. 2. Select one or more breakpoints that you want to remove. 3. Click the Remove Selected Breakpoints icon  on the toolbar (or right-click the selected breakpoints and then click  Remove).
Remove all breakpoints simultaneously in the Breakpoints view:	<ol style="list-style-type: none"> 1. Click the Breakpoints tab to show the Breakpoints view. 2. Click the Remove All Breakpoints icon  on the toolbar (or right-click any breakpoint and then click  Remove All).

Stepping through flow instances

This section tells you how to step through flow instances. It includes the following topics:

- “Resuming flow instance execution”
- “Running to completion”
- “Stepping over nodes” on page 467
- “Stepping into subflows” on page 467
- “Stepping out of subflows” on page 467
- “Stepping over source code” on page 468
- “Stepping into source code” on page 469
- “Stepping out of source code” on page 469

Resuming flow instance execution



Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Putting a test message” on page 459

The flow being debugged must be paused at a breakpoint.

To resume the flow execution:

1. Switch to the Debug perspective.
2. In the Debug view, click the **Resume Flow Execution** icon  on the toolbar (or right-click the flow stack frame, then click  **Resume**).

Tip: In the Debug perspective, when you choose to resume the execution of a flow instance, the flow instance pauses at the next breakpoint. If there is no breakpoint at which the flow instance can pause, the flow instance runs to completion and is removed from the Debug view.

Running to completion



Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Putting a test message” on page 459

The flow being debugged must be paused at a breakpoint.

To run to completion:

1. Switch to the Debug perspective.
2. In the Debug view, click the **Run to completion** icon  on the toolbar (or right-click the flow stack frame, then click  **Run to completion**). The flow

instance ignores all breakpoints while running to the end. The flow instance is automatically removed from the Debug view.

Tip: If you want the flow to carry on running, but you want to stop at the next breakpoint instead of running the flow to completion, see “Resuming flow instance execution” on page 466.

Stepping over nodes



Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Putting a test message” on page 459

The flow being debugged must be paused at a breakpoint.

To step over a node:

1. Switch to the Debug perspective.
2. In the Debug view, click the **Step Over Node** icon  on the toolbar (or right-click the flow stack frame, then click  **Step Over**). The debugger automatically pauses the message flow at the next place where a breakpoint either exists or can be added.

Stepping into subflows



Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Putting a test message” on page 459.

The flow being debugged must be paused at a breakpoint.

To step into a subflow:

1. Switch to the Debug perspective.
2. In the Debug view, click the **Step Into Subflow** icon  on the toolbar (or right-click the flow stack frame, then click  **Step Into**). The subflow opens in the Message Flow editor and displaces the parent message flow.

Stepping out of subflows



Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Putting a test message” on page 459
- “Stepping into subflows”

The flow being debugged must be paused at a breakpoint and you must have stepped into a subflow.

To step out of a subflow:

1. Switch to the Debug perspective.
2. In the Debug view, click the **Step Out of Subflow** icon  on the toolbar (or right-click the subflow stack frame, then click  **Step Out**). The debugger pauses the flow at the next breakpoint in the subflow, if there is one. If there are no remaining breakpoints in the subflow, the parent flow opens in the Message Flow editor and displaces the subflow. The parent flow is paused at the connection from the output terminal of the subflow.

Stepping over source code

Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Putting a test message” on page 459

The flow being debugged must be paused at a breakpoint containing ESQL code, or Java code, or mappings.

In a message flow there are three types of node that can contain source code:



- Nodes that contain ESQL code: Compute node, Filter node, or Database node that contains ESQL code.
- Nodes that contain Java code: User-defined node containing Java code, JavaCompute node.
- Nodes used to access and maintain mappings: a Mapping node, DataInsert node, DataUpdate node, DataDelete node, Extract node, or Warehouse node that contains mapping routines.

Note: mapping routines are implemented in ESQL - you can choose either to step through the ESQL code, or step through the mappings as described in “Debugging mappings” on page 472.

When you stop at a breakpoint at a piece of source code, you can choose how to step past this code:

- Step into the code, as described in “Stepping into source code” on page 469.
- Step over (that is, step through) the code, as described in this topic.
- After stepping into the code, step out of it, as described in “Stepping out of source code” on page 469.

To step over source code:

1. Switch to the Debug Perspective.
2. In the Debug view, click the **Step Over** icon  on the toolbar (or right-click the flow stack frame, then click  **Step Over**). The source code runs and the flow pauses at the next line of code. If the debugger is stepping through the last line of code, it automatically pauses at the next location where a breakpoint exists or can be added.

If you want to step out of the source code, see “Stepping out of source code.”

Stepping into source code

Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Putting a test message” on page 459

The flow being debugged must be paused at a breakpoint before a node containing ESQL code or Java code.

In a message flow there are three types of node that can contain source code:



- Nodes that contain ESQL code: Compute node, Filter node, or Database node that contains ESQL code.
- Nodes that contain Java code: User-defined node containing Java code, JavaCompute node.
- Nodes used to access and maintain mappings: a Mapping node, DataInsert node, DataUpdate node, DataDelete node, Extract node, or Warehouse node that contains mapping routines.

Note: mapping routines are implemented in ESQL - you can choose either to step through the ESQL code, or step through the mappings as described in “Debugging mappings” on page 472.

When you stop at a breakpoint just before one of the above types of node, you can choose how to step past the code:

- Step into the code, as described in this topic.
- Step over (that is, step through) the code, as described in “Stepping over source code” on page 468.
- After stepping into the code, step out of it, as described in “Stepping out of source code.”

To step into source code:

1. Switch to the Debug perspective.
2. In the Debug view, click the **Step into Source Code** icon  on the toolbar (or right-click the flow stack frame, then click  **Step Into**). You can now step over (that is, step through) the source code, line by line, in an environment that is optimized for source code debugging.

Note: If you want to step out of the source code after you have finished stepping through it, see “Stepping out of source code.”

Stepping out of source code

Before you start

To complete this task, you must have completed the following tasks:

- “Attaching to the flow engine” on page 456

- “Adding breakpoints” on page 462
- “Putting a test message” on page 459

The flow being debugged must be paused at a breakpoint containing ESQL code or Java code, and you must have stepped into the code.

In a message flow there are three types of node that can contain source code:



- Nodes that contain ESQL code: Compute node, Filter node, or Database node that contains ESQL code.
- Nodes that contain Java code: User-defined node containing Java code, JavaCompute node.
- Nodes used to access and maintain mappings: a Mapping node, DataInsert node, DataUpdate node, DataDelete node, Extract node, or Warehouse node that contains mapping routines.

Note: mapping routines are implemented in ESQL - you can choose either to step through the ESQL code, or step through the mappings as described in “Debugging mappings” on page 472.

When you stop at a breakpoint at a piece of source code, you can choose how to step past the code:

- Step into the code, as described in “Stepping into source code” on page 469.
- Step over (step through) the code, as described in “Stepping over source code” on page 468.
- After stepping into the code, step out of it, as described in this topic.

To step out of source code:

1. Switch to the Debug perspective.
2. In the Debug view, click the **Step Return** icon  on the toolbar (or right-click the flow stack frame, then click  **Step Return**). The source code runs, but the flow pauses at the next breakpoint within the source code, if there is one. If there is no breakpoint, or if the debugger is stepping out of the last line of code, the flow automatically pauses at the next location where a breakpoint either exists or can be added.

Debugging data

This section tells you how to debug data. It includes the following topics:

- “Debugging messages”
- “Debugging ESQL” on page 471
- “Debugging Java” on page 472
- “Debugging mappings” on page 472

Debugging messages

Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Putting a test message” on page 459

In the Variables view of the Debug perspective, you can view or alter messages currently traveling through the flow.

To view or alter a message:

1. Switch to the Debug perspective.
2. View the messages in the Variables view.

Tip: The Breakpoints view and the Variables view share the same pane. Click the tab at the bottom to select the view that you want.

3. To alter a message, right-click it and select an option from the menu.

Debugging ESQL

Before you start


Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Stepping into subflows” on page 467
- Step into a node with ESQL code (for example, a Compute, Filter, or Database node).

While stepping over the source code in an ESQL node (see “Stepping into source code” on page 469), the ESQL variables are displayed in the Variables view in the Debug Perspective. You can browse them and change their associated data values.

You can also set breakpoints on lines in the ESQL code.

To work with ESQL variables:

1. Switch to the Debug perspective.
2. Open the Variables view. Variables are shown in a tree, using the symbol .
3. To work with a variable, right-click it and select an option from the pop-up menu.

To use breakpoints on ESQL code lines:

1. Switch to the Debug perspective.
2. Open the ESQL editor.
3. Right-click a line where a breakpoint can be set. You cannot set a breakpoint on a comment line or a blank line.
4. Select from the menu to create, delete, or restrict the breakpoint, in a similar way to normal debugger breakpoints, as described in “Working with breakpoints” on page 462.

To debug promoted ESQL:

If you are debugging promoted ESQL, ensure that the Default Compatibility Level is correct by clicking **Window** → **Preferences** → **ESQL and Mapping** → **Code Generation** → **Default Compatibility Level**.

- If you are deploying to Version 5.0 brokers, select **5.0** from the drop-down list (this is the default setting).
- If you are deploying to Version 2.1 brokers, select **2.1** from the drop-down list.

If you change this setting, you must rebuild every message flow project in your workspace.

If you try to deploy a message set that uses functionality that is not supported by your level of broker, you will receive an error message. To solve this, modify your message set to ensure that it only uses functionality supported by your level of broker, or upgrade your broker to the required level.

Debugging Java


Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- Step into a node containing Java code (either a JavaCompute node, or a user-defined node with Java code).

While stepping over the Java code in a node (see “Stepping into source code” on page 469), the Java variables are displayed in the Variables view in the Debug perspective. You can browse them and change their associated data values.


To work with Java variables:

1. Switch to the Debug perspective.
2. Open the Variables view. Variables are shown in a tree, using the symbol .
3. To work with a variable, right-click it and select an option from the menu.

Debugging mappings



Before you start

To complete this task, you must have completed the following tasks:

- “Attaching to the flow engine” on page 456
- “Working with breakpoints” on page 462 (Set a breakpoint at the connection right before the mapping node. This breakpoint appears with the **Step into Source Code**  icon when a message is sent to the input queue.)

This topic describes how to debug a flow with mappings.

To check the mapping routines:

1. Switch to the Debug perspective.
2. In the Debug view, click the **Step into Source Code**  icon on the toolbar (or right-click on the flow stack frame and then click  Step Into). The Message Mapping editor opens with the mapping routine highlighted in both the Mapping editor and the Outline view.
3. To use breakpoints on mapping lines:
 - a. In the Message Mapping Editor, select the line for the mapping command you want to use, right-click on the space beside it and select from the menu to add or disable a breakpoint. (Alternatively double-click the same space to add or remove a breakpoint). You cannot set a breakpoint on a comment line or a blank line.
 - b. Select from the menu to create, delete, or restrict the breakpoint, in a similar way to normal debugger breakpoints, as described in “Working with breakpoints” on page 462.

4. Check the mapping routines by performing the various stepping actions (step into, step over, and step return). While you are doing this, in the Debug view, the stack frame shows the list of mapping commands and the current command. The Variables view shows your user-defined mapping variables, along with the input message (that is, the Debug Message). You can alter the user-defined variables.

When you have finished debugging the mapping, the message flow ends in the usual manner.

Managing flows and flow instances

This section tells you how to manage flows and flow instances. It includes the following topics:

- “Querying a broker for deployed flows”
- “Redeploying a flow”
- “Terminating a flow instance” on page 474
- “Detaching from the flow engine” on page 474

Querying a broker for deployed flows




Before you start

Complete the following task:

- “Attaching to the flow engine” on page 456

In the Debug perspective you can query a flow engine to find out what flows are currently deployed on it. This is useful when, for example, you want to access a flow that was not previously deployed, because the flow did not exist or because the flow was already being accessed by another developer.

To query a flow engine for deployed flows:

1. Switch to the Debug perspective.
2. In the Debug view, select the flow engine  that you want to query.
3. Click the **Refresh Selected Flow Engine to Get More Flow Types** icon  on the toolbar (or right-click the flow engine, then click  **Refresh**). The Debug view is refreshed with the names of the flows that are currently deployed and available on the flow engine.

Redeploying a flow



Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Putting a test message” on page 459

This topic tells you how to edit a flow to fix a problem that you have discovered while you are debugging and how to redeploy the flow.

To redeploy the flow:

1. Switch to the Debug perspective.
2. Detach the debugger from the flow engine by clicking the **Detach from the Selected Flow Engines** icon  on the toolbar.
3. Switch to the Broker Application Development perspective.
4. Edit the flow in the Message Flow editor and save your changes.
5. Switch to the Broker Administration perspective.
6. Double-click the bar file (Broker Archive) that contains your flow, to edit it. Remove the flow, then add it again, and save your changes. See “Adding files to a broker archive” on page 432 for more details.
7. Deploy your bar file by dragging it from the Broker Administration Navigator view and dropping it onto the execution group in the Domains view. Check the event log for messages to make sure that the deploy was successful.
8. Switch to the Debug perspective.
9. Click the down-arrow on the **Debug** icon  on the toolbar, and select **Debug** to invoke the **Debug (Create, manage, and run configurations)** wizard, and proceed as described in “Attaching to the flow engine” on page 456.

Terminating a flow instance



Before you start

Complete the following tasks:

- “Attaching to the flow engine” on page 456
- “Adding breakpoints” on page 462
- “Putting a test message” on page 459

While you are debugging, you might need to terminate a flow instance. For example, you might want to correct an error in your flow or source code. To do this, you must redeploy the flow correctly (see “Redeploying a flow” on page 473). However, you cannot redeploy the flow until you have terminated the flow instance.

To terminate the flow instance:

1. Switch to the Debug perspective.
2. In the Debug view, click the **Run to completion** icon  on the toolbar (or right-click the flow stack frame, then click  **Run to Completion**). The flow instance ignores all breakpoints while running to the end. The flow instance is automatically removed from the Debug view.

For more details, see “Running to completion” on page 466.

Detaching from the flow engine



Before you start

Complete the following task:

- “Attaching to the flow engine” on page 456

When you have finished debugging a flow in the Debug perspective, detach the flow debugger from the flow engine. This gives other developers an opportunity to attach to the flow engine and enhances the performance of your workbench environment.

To detach from the flow engine:

1. Switch to the Debug perspective.
2. In the Debug view, select the name of the flow engine from which you want to detach.
3. Click the **Detach from the Selected Flow Engines** icon  on the toolbar (or right-click the flow engine, then click  **Detach**). Any existing flow instances are automatically run to completion and the flow debugger detaches from the flow engine.

Part 4. Reference

Message flows	479	User database connections	700
Message flow preferences	479	Listing database connections that the broker	
Description properties for a message flow	479	holds	701
Guidance for defining keywords	480	Quiescing a database	701
Built-in nodes	482	User database DBCS restrictions and UNICODE	
AggregateControl node	483	support	702
AggregateReply node	485	Data integrity within message flows	702
AggregateRequest node	488	Validation properties for messages in the MRM	
Check node	490	domain	703
Compute node	492	Validation tab properties	703
Database node	500	Parsing on demand	706
DataDelete node	504	Exception list structure	707
DataInsert node	507	Database exception trace output	709
DataUpdate node	510	Conversion exception trace output	711
Extract node	513	Parser exception trace output	713
Filter node	515	User exception trace output	713
FlowOrder node	519	Configurable message flow properties	715
HTTPInput node	521	Message flow porting considerations	716
HTTPReply node	527	Message flow accounting and statistics data	717
HTTPRequest node	529	Message flow accounting and statistics details	717
Input node	540	Message flow accounting and statistics output	
JavaCompute node	542	formats	718
JMSInput node	546	Example message flow accounting and statistics	
JMSMQTransform node	557	data	730
JMSOutput node	558	Coordinated message flows	734
Label node	565	Database connections for coordinated message	
Mapping node	567	flows	734
MQeInput node	573	Database support for coordinated message flows	734
MQeOutput node	580	Element definitions for message parsers	735
MQGet node	584	Data types of fields and elements	735
MQInput node	593	The MQCFH parser	741
MQJMSTransform node	602	The MQCIH parser	742
MQOptimizedFlow node	603	The MQDLH parser	743
MQOutput node	605	The MQIIH parser	743
MQReply node	612	The MQMD parser	744
Output node	615	The MQMDE parser	745
Passthrough node	617	The MQRFH parser	745
Publication node	619	The MQRFH2 parser	746
Real-timeInput node	621	The MQRMH parser	746
Real-timeOptimizedFlow node	623	The MQSAPH parser	747
ResetContentDescriptor node	626	The MQWIH parser	747
RouteToLabel node	630	The SMQ_BMH parser	747
SCADAInput node	632	Message mappings	748
SCADAOutput node	639	Message Mapping editor	748
Throw node	642	Mapping node	756
TimeoutControl node	644	Migration of message mappings from Version	
TimeoutNotification node	647	5.0	759
Trace node	652	Restrictions on migrating message mappings	760
TryCatch node	656	XML constructs	762
Validate node	658	Example XML message	763
Warehouse node	661	The XML declaration	763
XMLTransformation node	665	The XML message body	765
User-defined nodes	671	XML document type declaration	769
Supported code pages	671	Data sources on z/OS	778
Chinese code page GB18030	699		
WebSphere MQ connections	699		
		ESQL reference	779

Syntax diagrams: available types	780	ESQL datetime functions	897
ESQL data types in message flows	780	ESQL numeric functions	902
ESQL BOOLEAN data type	780	ESQL string manipulation functions	915
ESQL datetime data types	780	ESQL field functions	925
ESQL NULL data type	786	ESQL list functions	936
ESQL numeric data types	786	Complex ESQL functions	938
ESQL REFERENCE data type	789	Miscellaneous ESQL functions	979
ESQL string data types	789	Broker properties accessible from ESQL and Java	983
ESQL-to-Java data-type mapping table	790	Special characters, case sensitivity, and comments	
ESQL variables	791	in ESQL	986
ESQL field references	792	ESQL reserved keywords	988
Namespaces	793	ESQL non-reserved keywords	988
Indexes	794	Example message	991
Types	794		
Summary	795	Flow application debugger	993
Target field references	796	Java Debugger	993
The effect of setting a field to NULL	797	Flow debugger shortcuts	993
ESQL operators	798	Debug view	993
ESQL simple comparison operators	798	Breakpoints view	994
ESQL complex comparison operators	799	Flow Breakpoint Properties dialog	994
ESQL logical operators	802	Variables view	994
ESQL numeric operators	803	Flow debugger icons and symbols	994
ESQL string operator	804	Debug perspective	994
Rules for ESQL operator precedence	804	Debug view	995
ESQL statements	804	Message Flow editor	995
ATTACH statement	806	Breakpoints view	996
BEGIN ... END statement	807	Variables view	996
BROKER SCHEMA statement	810		
CALL statement	813		
CASE statement	816		
CREATE statement	818		
CREATE FUNCTION statement	826		
CREATE MODULE statement	835		
CREATE PROCEDURE statement	837		
DECLARE statement	851		
DECLARE HANDLER statement	856		
DELETE FROM statement	857		
DELETE statement	860		
DETACH statement	860		
EVAL statement	861		
FOR statement	862		
IF statement	863		
INSERT statement	864		
ITERATE statement	866		
LEAVE statement	866		
LOG statement	867		
LOOP statement	869		
MOVE statement	870		
PASSTHRU statement	872		
PROPAGATE statement	874		
REPEAT statement	877		
RESIGNAL statement	878		
RETURN statement	878		
SET statement	880		
THROW statement	884		
UPDATE statement	885		
WHILE statement	888		
ESQL functions: reference material, organized by			
function type	889		
Calling ESQL functions	892		
ESQL database state functions	892		

Message flows

Message flow reference information is available for:

- “Message flow preferences”
- “Description properties for a message flow”
- “Built-in nodes” on page 482
- “User-defined nodes” on page 671
- “Supported code pages” on page 671
- “WebSphere MQ connections” on page 699
- “User database connections” on page 700
- “User database DBCS restrictions and UNICODE support” on page 702
- “Data integrity within message flows” on page 702
- “Validation properties for messages in the MRM domain” on page 703
- “Exception list structure” on page 707
- “Configurable message flow properties” on page 715
- “Message flow porting considerations” on page 716
- “Message flow accounting and statistics data” on page 717
- “Coordinated message flows” on page 734
- “Element definitions for message parsers” on page 735
- “Developing message mappings” on page 302
- “XML constructs” on page 762
- “Data sources on z/OS” on page 778

Message flow preferences

You can set Message flow preferences from **Window** → **Preferences** then click **Message Flow** in the left pane.

Property	Type	Meaning
Default version tag	String	Provide the default version information you would like to be set in the message flow Version property when you create a new message flow.

Description properties for a message flow

Property	Type	Meaning
Version	String	You can enter a version for the message flow in this field. This allows the version of the message flow to be displayed using the Eclipse properties view. A default for this field can be set in the messages flow preferences.
Short Description	String	You can enter a short description of the message flow in this field.

Property	Type	Meaning
Long Description	String	<p>You can add information to enhance the understanding of the message flow's function in this field.</p> <p>It is a string field and any standard alphanumeric characters can be used.</p> <p>You can also use this field to define a keyword and its value that will display for the deployed message flow in the properties view of Eclipse. An example is: \$MQSI Author=Fred MQSI\$</p> <p>When the properties of the deployed message flow are displayed, this will add a row to the display showing "Author" as the property name and "Fred" as its value.</p> <p>For information on keywords see "Guidance for defining keywords."</p>

To view and edit the properties of a message flow click **Flow** → **Properties**.

Guidance for defining keywords

This topic contains the rules to follow when defining keywords. Keywords and their values are displayed in the properties view of a deployed object.

A number of objects in WebSphere Message Broker can have additional information added to the object. This information can display information about an object after the object has been deployed. The default information that is displayed is the time the object was deployed and the last time the object was modified. Version information can be specifically set using the **Version** property for message sets and message flows.

You can define custom keywords and their values that the Configuration Manager will interpret as additional information to be displayed in the properties view. For example, you can define keywords for "Author" and "Subflow 1 Version":

```
$MQSI Author=John Smith MQSI$
$MQSI Subflow 1 Version=v1.3.2 MQSI$
```

The information the Configuration Manager shows is:

Object name	
Deployment Time	28-Aug-2004 15:04
Modification Time	28-Aug-2004 14:27
Version	v1.0
Author	John Smith
Subflow 1 Version	v1.3.2

In this display the version information has also been defined using the Version property of the object. If the version information had not been defined using the property, it would be omitted from this display.

The syntax for defining a keyword and its associated value is:

```
$MQSI KeywordName = KeywordValue MQSI$
```


Where:

\$MQSI

\$MQSI opens the definition. It can be followed by an optional underscore or white space character that is ignored.

KeywordName

The name of the keyword that you are setting the value for. It can be made up of any sequence of alphanumeric characters apart from the equals (=) sign. It can contain white space characters, but any leading or trailing white space characters will be omitted.

= The equals (=) sign is the delimiter between the keyword and the value that you are setting it to.

KeywordValue

The value that the keyword will be set to. It can be made up of any sequence of alphanumeric characters. It can contain white space characters, but any leading or trailing white space characters will be omitted.

MQSI\$

MQSI\$ closes the keyword definition.

Examples

Example definitions	Interpreted keyword and value	Comments
\$MQSI Author=JohnMQSI\$ or \$MQSI Author=John MQSI\$ or \$MQSI Author = John MQSI\$	Keyword = "Author" Value = "John"	Each of these is a basic example of what can be set and shows that the leading and trailing white space characters for the name and value parameters is ignored.
\$MQSI_Author = John MQSI\$	Keyword = "Author" Value = "John"	The first character after \$MQSI can be an underscore character. The underscore character is omitted in the interpreted keyword. If a second underscore character appears, this will form part of the keyword name.
\$MQSI Flow designer = John Smith MQSI\$	Keyword = "Flow designer" Value = "John Smith"	White space characters are accepted for each parameter value.
\$MQSI bar = MQSI\$	Keyword = "bar" Value = ""	The keyword value can be set to an empty ("") string.
\$MQSI_mqsitag=\$MQSI\$MQSI\$	Keyword = "mqsitag" Value = "\$"	This is a poorly formatted definition. After defining the keyword name the parser is looking to find the delimiters that form the boundary of the value to be set. In this case the only character prior to the MQSI\$ that closes the definition is a '\$' and that is set as the keyword value.
\$MQSI=barMQSI\$		This pattern is ignored because the keyword name cannot be an empty string.
\$MQSItagbarMQSI\$		This pattern is ignored because there is not a separator (=) between the keyword name and the keyword value.

Built-in nodes

WebSphere Message Broker supplies built-in nodes that you can use to define your message flows. For information about each of these nodes, follow the appropriate link below. The nodes listed here are grouped according to the function that they provide.

Input and output

- [“MQInput node” on page 593](#)
- [“MQOptimizedFlow node” on page 603](#)
- [“MQOutput node” on page 605](#)
- [“MQGet node” on page 584](#)
- [“MQReply node” on page 612](#)
- [“Publication node” on page 619](#)
- [“MQeInput node” on page 573](#)
- [“MQeOutput node” on page 580](#)
- [“SCADAInput node” on page 632](#)
- [“SCADAOutput node” on page 639](#)
- [“HTTPInput node” on page 521](#)
- [“HTTPReply node” on page 527](#)
- [“HTTPRequest node” on page 529](#)
- [“Real-timeInput node” on page 621](#)
- [“Real-timeOptimizedFlow node” on page 623](#)
- [“Input node” on page 540](#)
- [“Output node” on page 615](#)
- [“JMSInput node” on page 546](#)
- [“JMSOutput node” on page 558](#)

Message manipulation and transformation

- [“Compute node” on page 492](#)
- [“Database node” on page 500](#)
- [“DataDelete node” on page 504](#)
- [“DataInsert node” on page 507](#)
- [“DataUpdate node” on page 510](#)
- [“Extract node” on page 513](#)
- [“JavaCompute node” on page 542](#)
- [“JMSMQTransform node” on page 557](#)
- [“MQJMSTransform node” on page 602](#)
- [“Mapping node” on page 567](#)
- [“Warehouse node” on page 661](#)
- [“XMLTransformation node” on page 665](#)

Collating requests

- [“AggregateControl node” on page 483](#)
- [“AggregateReply node” on page 485](#)
- [“AggregateRequest node” on page 488](#)

Decision making

- [“Check node” on page 490](#)
- [“Filter node” on page 515](#)
- [“FlowOrder node” on page 519](#)
- [“Label node” on page 565](#)
- [“ResetContentDescriptor node” on page 626](#)
- [“RouteToLabel node” on page 630](#)
- [“TimeoutControl node” on page 644](#)

- “TimeoutNotification node” on page 647
- “Validate node” on page 658

Subflow identification

- “Passthrough node” on page 617

Error handling and reporting

- “Throw node” on page 642
- “Trace node” on page 652
- “TryCatch node” on page 656

AggregateControl node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the AggregateControl node” on page 484
- “Terminals and properties” on page 484

Purpose

Use the AggregateControl node to mark the beginning of a fan-out of requests that are part of an aggregation.

Aggregation is an extension of the request/reply application model. It combines the generation and fan-out of a number of related requests with the fan-in of the corresponding replies, and compiles those replies into a single aggregated reply message.

The aggregation function is provided by the following three nodes:

1. The AggregateControl node marks the beginning of a fan-out of requests that are part of an aggregation. It sends a control message that is used by the AggregateReply node to match the different requests that have been made. The information propagated from the control terminal includes the broker identifier, the aggregate name property, and the timeout property. The aggregation information that is added to the message Environment by the AggregateControl node must not be changed.
2. The AggregateRequest node records the fact that the request messages have been sent. It also collects information that helps the AggregateReply node to construct the aggregated reply message. The information added to the message Environment by the AggregateRequest must be preserved otherwise the aggregation will fail.
3. The AggregateReply node marks the end of an aggregation fan-in. It collects replies and combines them into a single aggregated reply message.

The AggregateControl node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following samples to see how you can use this node:

- Aggregation sample

- Airline Reservations sample

sample to see how you can use this node:

Configuring the AggregateControl node

When you have put an instance of the AggregateControl node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the AggregateControl node as follows:

1. Enter a value for the *Aggregate Name*. This name is used to associate the fan-out message flow with the fan-in message flow. This value must be contextually unique within a broker. This property is mandatory; you must enter a value.
2. Enter a *Timeout*. This value is specified in seconds. You must enter a value (or accept the initial value shown, 0) because this property is mandatory. If you accept the value, the timeout is disabled for fan-outs from this node (that is, replies are waited for indefinitely). If not all responses are received, the message flow continues to wait, and does not complete. Therefore, you are recommended to set a value greater than 0. Refer to the "AggregateReply node" on page 485 for further information about timeouts.
3. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
4. Click **Apply** to make the changes to the AggregateControl node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The AggregateControl node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Out	The output terminal to which the original message is routed when processing completes successfully.
Control	The output terminal to which a control message is routed. The control message is sent to a corresponding AggregateReply node. Note: The Control terminal is deprecated in Version 6.0, to use connections from the Control terminal see, "Using control messages in aggregation flows" on page 397.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The AggregateControl node Basic properties are described in the following table.

Property	M	C	Default	Description
Aggregate Name	Yes	Yes		A name that can be used to associate the fan-out message flow with the fan-in message flow.
Timeout (secs)	Yes	No	0	The amount of time, in seconds, for which replies are waited to arrive at the fan-in.

Note: On z/OS if the Timeout property is not set to zero, set the queue manager parameter **EXPRYINT** to 5.

The AggregateControl node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

AggregateReply node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 486
- “Configuring the AggregateReply node” on page 486
- “Terminals and properties” on page 487

Purpose

Use the AggregateReply node to mark the end of an aggregation fan-in. This node collects replies and combines them into a single compound message.

Aggregation is an extension of the request/reply application model. It combines the generation and fan-out of a number of related requests with the fan-in of the corresponding replies, and compiles those replies into a single aggregated reply message.

The aggregation function is provided by the following three nodes:

1. The AggregateControl node marks the beginning of a fan-out of requests that are part of an aggregation. It sends a control message that is used by the AggregateReply node to match the different requests that have been made. The information that is propagated from the control terminal includes the broker identifier, the aggregate name property, and the timeout property. The aggregation information that is added to the message Environment by the AggregateControl node must not be changed.
2. The AggregateRequest node records the fact that the request messages have been sent. It also collects information that helps the AggregateReply node to construct the aggregated reply message. The information that is added to the message Environment by the AggregateRequest must be preserved; otherwise the aggregation fails.
3. The AggregateReply node marks the end of an aggregation fan-in. It collects replies and combines them into a single aggregated reply message.

The AggregateReply node is represented in the workbench by the following icon:



When incoming messages are stored by the `AggregateReply` node before all responses for the aggregation are received, the persistence of the message determines whether the message survives a restart.

If during an aggregation, one or more of the response messages are not received by the `AggregateReply` node, the normal timeout or unknown message processing deals with the responses that have already been received.

Using this node in a message flow

Look at the following samples to see how you can use this node:

- Aggregation sample
- Airline Reservations sample

Configuring the `AggregateReply` node

When you have put an instance of the `AggregateReply` node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the `AggregateReply` node as follows:

1. Enter a value for the *Aggregate Name*. This name is used to associate the fan-in message flow with the fan-out message flow. This value must be contextually unique within a broker. This property is mandatory; you must enter a value.
2. Enter a value for *Unknown Message Timeout*. This value is specified in seconds. It specifies the amount of time for which messages that cannot be identified as valid replies are held before being propagated to the unknown terminal.
If you enter 0, or do not enter a value, the timeout is disabled and unknown messages are propagated to the unknown terminal upon receipt.
3. Select the *Transaction Mode* to define the transactional characteristics of this message:
 - If you select the check box, the subsequent message flow is under transaction control. This remains true for messages that derive from the output message and are output by an `MQOutput` node, unless the `MQOutput` node explicitly overrides the transaction status. This is the default. (No other node can change the transactional characteristics of the output message.)
 - If you clear the check box, the subsequent message flow is not under transaction control. This remains true for messages that derive from the output message and are output by an `MQOutput` node, unless the `MQOutput` node has specified that the message should be put under syncpoint.
4. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
5. Click **Apply** to make the changes to the `AggregateReply` node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The AggregateReply node terminals are described in the following table.

Terminal	Description
Control	The input terminal that accepts control messages sent by a corresponding AggregateControl node. Note: The Control terminal is deprecated in Version 6.0; to use connections to the Control terminal see “Using control messages in aggregation flows” on page 397.
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if a failure is detected during processing.
Unknown	The output terminal to which messages are routed when they cannot be identified as valid reply messages.
Out	The output terminal to which the compound message is routed when processing completes successfully.
Timeout	The output terminal to which the incomplete compound message is routed when the timeout interval that is specified in the corresponding AggregateControl node has expired.
Catch	The output terminal to which the message is routed if an exception is thrown downstream and then caught by this node.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The AggregateReply node Basic properties are described in the following table.

Property	M	C	Default	Description
Aggregate Name	Yes	Yes		A name that can be used to associate the fan-in message flow with the fan-out message flow. This property is mandatory.
Unknown Message Timeout	No	No	0	The amount of time for which messages that cannot be identified as replies are held before being propagated to the unknown terminal.
Transaction Mode	Yes	No	Selected	Whether messages propagated by this node are put transactionally. If you select the check box, this action is performed.

Note: On z/OS, if the Unknown Message Timeout property is not set to zero, set the queue manager parameter **EXPRYINT** to 5.

The Description properties of the AggregateReply node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.

Property	M	C	Default	Description
Long Description	No	No		Text that describes the purpose of the node in the message flow.

AggregateRequest node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the Aggregate Request node” on page 489
- “Terminals and properties” on page 489

Purpose

Use the AggregateRequest node to record the fact that request messages have been sent. This node also collects information that helps the AggregateReply node to construct the compound message.

Aggregation is an extension of the request/reply application model. It combines the generation and fan-out of a number of related requests with the fan-in of the corresponding replies, and compiles those replies into a single aggregated reply message.

The aggregation function is provided by the following three nodes:

1. The AggregateControl node marks the beginning of a fan-out of requests that are part of an aggregation. It sends a control message that is used by the AggregateReply node to match the different requests that have been made. The information propagated from the control terminal includes the broker identifier, the aggregate name property, and the timeout property. The aggregation information that is added to the message Environment by the AggregateControl node must not be changed.
2. The AggregateRequest node records the fact that the request messages have been sent. It also collects information that helps the AggregateReply node to construct the aggregated reply message. The information added to the message Environment by the AggregateRequest must be preserved otherwise the aggregation will fail.
3. The AggregateReply node marks the end of an aggregation fan-in. It collects replies and combines them into a single aggregated reply message.

The AggregateRequest node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following samples to see how you can use this node:

- Aggregation sample
- Airline Reservations sample

Configuring the Aggregate Request node

When you have put an instance of the `AggregateRequest` node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the `AggregateRequest` node as follows:

1. Enter a value for the *Folder Name*. This name is used as a folder in the `AggregateReply` node's compound message to store the reply to this request. This property is mandatory; you must enter a value. The value does not have to be unique.
2. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
3. Click **Apply** to make the changes to the `AggregateRequest` node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The `AggregateRequest` node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts messages sent as part of an aggregate request.
Out	The output terminal to which the input message is routed when processing completes successfully.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The `AggregateRequest` node Basic property is described in the following table.

Property	M	C	Default	Description
Folder Name	Yes	No		The name that is used as a folder in the <code>AggregateReply</code> node's compound message. This property is mandatory.

The `AggregateRequest` node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Check node

Attention: The Check node is deprecated in WebSphere Message Broker Version 6.0. Although message flows that contain a Check node remain valid in WebSphere Message Broker Version 6.0, where possible redesign your message flows so that any Check node is replaced by a Validate node.

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the Check node” on page 491
- “Terminals and properties” on page 491

Purpose

Use the Check node to compare the structure of a message arriving on its input terminal with a message structure definition that you supply when you configure the Check node. The message structure definition comprises the message domain; if the message domain is MRM, you can also specify any combination of message set and message type. The Check node checks only the message structure definition; it does not check the message body.

The domain, set, and type of the message are collectively called the *message template*. The domain defines the parser that is used for the message. The set is the message set to which the message belongs. The type is the structure of the message itself. You can check the incoming message against one or more of these properties. The message property is checked only if you select its corresponding *Check* property, which means that a message property containing a null string can be compared.

If the message properties match the specification, the message is propagated through the match terminal of the node. If the message properties do not match the specification, the message is propagated through the failure output terminal. If the failure terminal is not connected to some failure handling processing, an exception is thrown.

The Check node is represented in the workbench by the following icon:



Using this node in a message flow

You can use the Check node to ensure that the message is routed appropriately through the message flow. For example, you can configure it to direct a message that requests stock purchases through a different route from that required for a message that requests stock sales.

Another example of this node’s use is the receipt of electronic messages from your staff at your head office. These messages are used for multiple purposes, for example to request technical support or stationery, or to advise you about new customer leads. These messages can be processed automatically because your staff fill in a standard form. If you want these messages to be processed separately from other messages received, use the Check node to ensure that only staff messages that have a specific message type are processed by this message flow.

Configuring the Check node

When you have put an instance of the Check node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Check node as follows:

1. To check the parser to be used for the incoming message, select the box *Check Domain* and enter one of the following values (in uppercase) in the *Domain* field:
 - MRM
 - XML
 - XMLNS
 - JMSMap
 - JMSStream
 - IDOC
 - MIME
 - BLOB

Use this option to check that a message belongs to a particular domain.

2. If you are using the MRM or IDOC parser, check that the incoming message belongs to a particular message set by selecting the box *Check Set* and entering the identifier of the message set in *Set*. This identifier can be found in the properties of the message set when you view it in the editor. WebSphere Message Broker generates the identifier when you create the message set; it is something like DHHJQC06U001. You must enter the identifier exactly as shown in the message set properties.

Leave *Set* clear for the XML, JMS, MIME, and BLOB parsers.

Use this option to check that a message belongs to a particular message set.

3. If you are using the MRM parser, check that the incoming message is a particular message type by selecting the box *Check Type* and entering the identifier of the message in *Type*.

This identifier can be found in the properties of the message when you view the message in the editor. You specify the message identifier when you create the message. You must enter the identifier exactly as shown in the message properties.

Leave *Type* clear for the XML, JMS, IDOC, MIME, and BLOB parsers.

Use this option to check that a message matches a particular definition.

4. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
5. Click **Apply** to make the changes to the Check node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Check node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if the incoming message does not match the specified properties.
Match	The output terminal to which the message is routed if the incoming message matches the specified properties.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Check node Basic properties are described in the following table.

Property	M	C	Default	Description
Domain	No	No		The name of the domain.
Check Domain	Yes	No	Cleared	Whether to check the incoming message against the Domain property. If you select the check box, this action is performed.
Set	No	No		The message set to which the incoming message belongs.
Check Set	Yes	No	Cleared	Whether to check the incoming message against the Set property. If you select the check box, this action is performed.
Type	No	No		The message identifier.
Check Type	Yes	No	Cleared	Whether to check the incoming message against the Type property. If you select the check box, this action is performed.

The Check node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Compute node

This topic contains the following sections:

- “Purpose” on page 493
- “Using this node in a message flow” on page 493
- “Configuring the Compute node” on page 493
- “Defining database interaction” on page 494
- “Specifying ESQL” on page 495
- “Setting the mode” on page 496
- “Validating messages” on page 497
- “Terminals and properties” on page 497

Purpose

Use the Compute node to construct one or more new output messages. These output messages might be created by modifying the information that is provided in the input message, or the output messages might be created using only new information which might (or might not) be taken from a database. Elements of the input message (for example, headers, header fields, and body data), its associated environment, and its exception list can be used to create the new output message.

You specify how the new messages are created by coding ESQL in the message flow ESQL resource file. You can both create and modify the components of the message using ESQL expressions, and can refer to elements of both the input message and data from an external database. An expression can use arithmetic operators, text operators (for example, concatenation), logical operators, and other built-in functions.

Use the Compute node to:

- Build a new message using a set of assignment statements
- Copy messages between parsers
- Convert messages from one code set to another
- Transform messages from one format to another

You define the ESQL statements in a module associated with this node in the ESQL (.esql) file associated with this message flow. You must create this file to complete the definition of the message flow.

The Compute node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following samples to see how you can use this node:

- Airline Reservations sample
- Aggregation sample
- JMS Nodes sample
- Large Messaging sample
- Message Routing sample
- Scribble sample
- Timeout Processing sample
- Video Rental sample

Consider a message flow in which you want to give each order that you receive a unique identifier for audit purposes. The Compute node does not modify its input message; it creates a new, modified copy of the message as an output message. You can use the Compute node to insert a unique identifier for your order into the output message, which can be used by subsequent nodes in the message flow.

Configuring the Compute node

When you have put an instance of the Compute node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Compute node by:

1. "Defining database interaction"
2. "Specifying ESQL" on page 495
3. "Setting the mode" on page 496
4. "Validating messages" on page 497

When you have completed your configuration, click **Apply**. This makes the changes to the Compute node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog. Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Defining database interaction:

If you want to access a database from this node:

- Specify in *Data Source* the name by which the appropriate database is known on the system on which this message flow is to execute. The broker connects to this database with user ID and password information that you have specified on the `mqsicreatebroker`, `mqsiexchangebroker`, or `mqsisetdbparms` command.
On z/OS systems, the broker uses the broker started task ID, or the user ID and password that were specified on the `mqsisetdbparms` command JCL, BIPsDBP in the customization data set <hlq>.SBIPPROC.
- Select the *Transaction* setting from the drop-down menu. The values are:
 - *Automatic* (the default). The message flow, of which the Compute node is a part, is committed if it is successful. That is, the actions that you define in the ESQL module are performed on the message and it continues through the message flow. If the message flow fails, it is rolled back. If you choose *Automatic*, the ability to commit or roll back the action of the Compute node on the database depends on the success or failure of the entire message flow.
 - *Commit*. If you want to commit the action of the Compute node on the database, irrespective of the success or failure of the message flow as a whole, select *Commit*. The database update is committed even if the message flow itself fails.

The value that you choose is implemented for the one or more database tables that you have added: you cannot select a different value for each table.

- Select *Basic* in the properties dialog navigator and select or clear the two check boxes:
 - If you want database warning messages to be treated as errors and want the node to propagate the output message to the failure terminal, select the *Treat warnings as errors* check box. The box is initially cleared.
When you select the box, the node handles all positive return codes from the database as errors and generates exceptions in the same way as it does for the negative, or more serious, errors.
If you do not select the box, the node treats warnings as normal return codes, and does not raise any exceptions. The most significant warning raised is not found, which can be handled as a normal return code safely in most circumstances.
 - If you want the broker to generate an exception when a database error is detected, select the *Throw exception on database error* check box. The box is initially selected.

If you clear the box, you must include ESQL to check for any database error that might be returned after each database call that you make (you can use `SQLCODE` and `SQLSTATE` to do this). If an error occurs, you must handle the error in the message flow to ensure the integrity of the broker and the database: the error is ignored if you do not handle it through your own processing, because you have chosen not to invoke the default error handling by the broker. For example, you can include the ESQL `THROW` statement to throw an exception in this node, or you can use the `Throw` node to generate your own exception at a later point in the message flow.

Specifying ESQL:

Code ESQL statements to customize the behavior of the `Compute` node. For example, you can customize it to create a new output message or messages, using input message or database content (unchanged or modified), or new data. For example, you might want to modify a value in the input message by adding a value from a database, and store the result in a field in the output message.

Code the ESQL statements that you want in an ESQL file associated with the message flow in which you have included this instance of the `Compute` node. The ESQL file, which by default has the name `<message_flow_name>.esql`, contains ESQL for every node in the message flow that requires it. Each portion of code that is related to a specific node is known as a module.

If an ESQL file does not already exist for this message flow, right-click the `Compute` node and click **Open ESQL**. This creates and opens a new ESQL file in the ESQL editor view.

If the file already exists, click the **Browse** button beside the *ESQL Module* property. This displays the `Module Selection` dialog, which lists the available `Compute` node modules defined in the ESQL files that are accessible by this message flow (ESQL files can be defined in other, dependent, projects). Select the appropriate module and click **OK**. If no suitable modules are available, the list is empty.

If the module that you have specified doesn't exist, it is created for you and the editor positions the file to display it. If the file and the module already exist, the editor positions the file and displays and highlights the correct module.

If you prefer, you can open the appropriate ESQL file in the `Resource Navigator` and select this node in the `Outline` view.

If a module skeleton is created for this node in a new or existing ESQL file, it consists of the following ESQL. The default module name is shown in this example:

```
CREATE COMPUTE MODULE <flow_name> Compute
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    -- CALL CopyMessageHeaders();
    -- CALL CopyEntireMessage();
    RETURN TRUE;
END;

CREATE PROCEDURE CopyMessageHeaders() BEGIN
    DECLARE I INTEGER 1;
    DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
    WHILE I < J DO
        SET OutputRoot.*[I] = InputRoot.*[I];
        SET I = I + 1;
    END DO;
END;
```

```

        END WHILE;
    END;

    CREATE PROCEDURE CopyEntireMessage() BEGIN
        SET OutputRoot = InputRoot;
    END;
END MODULE;

```

Note: If you want to deploy the message flow that contains this Compute node to a broker whose version is earlier than version 5.0, you must make the following changes to the ESQL in the module skeleton shown above:

- Replace


```

      DECLARE I INTEGER 1;
      
```

 by


```

      DECLARE I INTEGER; SET I=1;
      
```
- Replace


```

      DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
      
```

 by


```

      DECLARE J INTEGER; SET J=CARDINALITY(InputRoot.*[]);
      
```

If you create your own ESQL module, you must create this skeleton exactly as shown except for the procedure calls and definitions (described below). You can change the default name, but ensure that the name you specify matches the name of the corresponding node property *ESQL Module*. If you want the module name to include one or more spaces, enclose the name in double quotes in the *ESQL Module* property.

Add your own ESQL to customize this node after the BEGIN statement that follows CREATE FUNCTION, and before RETURN TRUE. You can use the two calls included in the skeleton, to procedures CopyEntireMessage and CopyMessageHeaders.

These procedures, defined following function Main, provide common functions that you might want when you manipulate messages. The calls in the skeleton are commented out; remove the comment markers if you want to use the procedure. If you do not want to use a procedure, remove both the call and the procedure definition from the module.

In previous releases, the functions performed by these procedures were provided by the two equivalent Compute node radio buttons on the properties dialog.

You can also create an ESQL file using the **File** → **New** → **Message Flow ESQL File**.

Setting the mode:

When you select the *Compute mode*, you specify whether the Message, LocalEnvironment (previously specified as DestinationList), and Exception List components, that are either generated within the node or contained within the incoming message, are used by default in the output message.

This default is used when the transformed message is routed to the Out terminal when processing in the node is completed. The default is also used whenever a PROPAGATE statement does not specify the composition of its output message.

Those components that are not included in your selection are passed on unchanged; even if you modify those components, the updates are local to this node.

The Environment component of the message tree is not affected by the mode setting. Its contents, if any, are passed on from this node in the output message.

You must set this property to correctly reflect the output message format that you require. If you select an option (or accept the default value) that does not include a particular part of the message, that part of the message is not included in any output message that is constructed.

The options are explained in the following table.

Mode	Description
Message (the default)	The message is generated or passed through by the Compute node as modified within the node.
LocalEnvironment	The LocalEnvironment tree structure is generated or passed through by the Compute node as modified within the node.
LocalEnvironment And Message	The LocalEnvironment tree structure and message are generated or passed through by the Compute node as modified by the node.
Exception	The Exception List is generated or passed through by the Compute node as modified by the node.
Exception And Message	The Exception List and message are generated or passed through by the Compute node as modified by the node.
Exception and LocalEnvironment	The Exception List and LocalEnvironment tree structure are generated or passed through by the Compute node as modified by the node.
All	The message, Exception List, and LocalEnvironment are generated or passed through by the Compute node as modified by the node.

Because the Compute node has both an input and output message, you can use ESQL to refer to fields in either. You can also work with both InputLocalEnvironment and OutputLocalEnvironment, and InputExceptionList and OutputExceptionList, as well as the input and output message bodies.

Validating messages:

Set the validation properties to define how the message that is produced by the Compute node is to be validated. Note that these properties do not cause the input message to be validated. It is expected that, if such validation is required, the validation has already been performed by the input node or a preceding validation node.

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

Terminals and properties

The Compute node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the input message is routed if an unhandled exception occurs during the computation.
Out	The output terminal to which the transformed message is routed when processing in the node is completed. The transformed message might also be routed to this terminal by a PROPAGATE statement.
Out1	The first alternate output terminal to which the transformed message might be routed by a PROPAGATE statement.
Out2	The second alternate output terminal to which the transformed message might be routed by a PROPAGATE statement.
Out3	The third alternate output terminal to which the transformed message might be routed by a PROPAGATE statement.
Out4	The fourth alternate output terminal to which the transformed message might be routed by a PROPAGATE statement.

Note: See “PROPAGATE statement” on page 874 for the syntax of the PROPAGATE statement.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Compute node Basic properties are described in the following table.

Property	M	C	Default	Description
Data Source	No	Yes		The ODBC data source name for the database within which reside any tables to which you refer in the ESQL file associated with this message flow (identified in the <i>ESQL Module</i> property). You can specify only one data source for the node.
Transaction	Yes	No	Automatic	The transaction mode for the node. This can be Automatic or Commit. It is valid only if you have selected a database table for input.
ESQL Module	Yes	No	Compute	The name of the module within the ESQL file that contains the statements to execute against the database and input and output messages.
Compute Mode	Yes	No	Message	Choose from: <ul style="list-style-type: none"> • Message • LocalEnvironment • LocalEnvironment And Message • Exception • Exception And Message • Exception And LocalEnvironment • All These are explained in “Setting the mode” on page 496.
Treat warnings as errors	Yes	No	Cleared	Treat database SQL warnings as errors. If you select the check box, this action is performed.

Property	M	C	Default	Description
Throw exception on database error	Yes	No	Selected	Database errors cause the broker to throw an exception. If you select the check box, this action is performed.

The Validation properties of the Compute node are described in the following table.

Property	M	C	Default	Description
Validate	Yes	Yes	None	Whether validation takes place. Valid values are None, Content and Value, Content, and Inherit.
Failure Action	Yes	No	Exception	What happens if a validation failure occurs. You can set this property only if <i>Validate</i> is set to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that all value constraints are included in the validation.
Fix	Yes	No	None	This property cannot be edited. Minimal fixing is provided. Valid values are None, and Full.

The properties of the General Message Options for the MQGet node are described in the following table.

Property	M	C	Default	Description
Parse Timing	Yes	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to "Parsing on demand" on page 706 for a full description of this property.
Use MQRFH2C Compact Parser for MQRFH2 Domain	No	No	False	This property controls whether the MQRFH2C Compact Parser, instead of the MQRFH2 parser, is used for MQRFH2 headers.

The XMLNSC parser properties for the Compute node are described in the following table.

Property	M	C	Default	Description
Use XMLNSC Compact Parser for XMLNS Domain	Yes	Cleared	No	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.
Mixed Content Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.

Property	M	C	Default	Description
Comments Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.
Processing Instructions Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The Compute node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Database node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 501
- “Configuring the Database node” on page 501
- “Terminals and properties” on page 503

Purpose

Use the Database node to interact with a database in the specified ODBC data source. You define the nature of the interaction by coding ESQL statements that specify the data from the input message, and perhaps transform it in some way (for example, to perform a calculation), and assign the result to a database table.

You can set a property to control whether the update to the database is committed immediately, or deferred until the message flow completes, at which time the update is committed or rolled back according to the overall completion status of the message flow.

Although you can use this node to update the database, you cannot make any updates to the message.

You can use specialized forms of this node to:

- Update values within a database table (the DataUpdate node)
- Insert rows into a database table (the DataInsert node)
- Delete rows from a database table (the DataDelete node)
- Store the message, or parts of the message, in a warehouse (the Warehouse node)

The Database node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following samples to see how you can use this node:

- Airline Reservations sample
- Error Handler sample

Consider a situation in which you receive an order for 20 monitors. If you have sufficient numbers of monitors in your warehouse, you want to decrement the stock level on your stock database. You can use the Database node to check that you have sufficient monitors available, and decrement the value of the quantity field in your database.

Configuring the Database node

When you have put an instance of the Database node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Database node as follows:

1. Specify in *Data Source* the name by which the appropriate database is known on the system on which this message flow is to execute. The broker connects to this database with user ID and password information that you have specified on the `mqsicreatebroker`, `mqsichangebroker`, or `mqsisetdbparms` command.

On z/OS systems, the broker uses the broker started task ID, or the user ID and password that were specified on the `mqsisetdbparms` command JCL, `BIPSDBP` in the customization data set `<hlq>.SBIPPROC`.

2. In *Statement*, identify the module within an ESQL file that contains the ESQL statements that are to be executed in this node. If you want the module name to include one or more spaces, enclose the name in double quotes in the *Statement* property.

When you code ESQL statements that interact with tables, those tables are assumed to exist within this database. If they do not, a database error is generated by the broker at runtime.

Code ESQL statements to customize the behavior of the Database node in an ESQL file that is associated with the message flow in which you have included this instance of the Database node. The ESQL file, which by default has the name `<message_flow_name>.esql`, contains ESQL for every node in the message flow that requires it. Each portion of code that is related to a specific node is known as a module.

If an ESQL file does not already exist for this message flow, right-click the Database node and click **Open ESQL**. This creates and opens a new ESQL file in the ESQL editor view.

If an ESQL file already exists, click the **Browse** button beside the *Statement* property. This displays the Module Selection dialog, which lists the available Database node modules defined in the ESQL files that are accessible by this message flow (ESQL files can be defined in other, dependent, projects). Select the appropriate module and click **OK**. If no suitable modules are available, the list is empty.

If the module that you have specified doesn't exist, it is created for you and the editor positions the file to display it. If the file and the module already exist, the editor positions the file and displays and highlights the correct module.

If you prefer, you can open the appropriate ESQL file in the Resource Navigator and select this node in the Outline view.

If a module skeleton is created for this node in a new or existing ESQL file, it consists of the following ESQL. The default module name is shown in this example:

```
CREATE DATABASE MODULE <flow_name>_Database
    CREATE FUNCTION Main() RETURNS BOOLEAN
    BEGIN
        RETURN TRUE;
    END;
END MODULE;
```

If you create your own ESQL module, create exactly this skeleton. You can update the default name, but ensure that the name that you specify matches the name of the corresponding node property Statement.

Add your own ESQL to customize this node after the BEGIN statement and before RETURN TRUE.

You can also create an ESQL file by clicking **File** → **New** → **Message Flow ESQL File**.

You can use all the ESQL statements including SET, WHILE, DECLARE, and IF in this module, but (unlike the Compute node) the Database node propagates unchanged the message that it receives at its input terminal to its output terminal. This means that, like the Filter node, you have only one message to refer to in a Database node.

Because you can't modify any part of any message, the assignment statement (the SET statement, not the SET clause of the INSERT statement) can assign values only to temporary variables. The scope of actions that you can take with an assignment statement is therefore limited.

If you prefer, you can open the appropriate ESQL file in the Resource Navigator and select this node in the Outline view.

3. Select the *Transaction* setting from the drop-down menu. The values are:
 - **Automatic** (the default). The message flow, of which the Database node is a part, is committed if it is successful. That is, the actions that you define in the ESQL module are performed and the message continues through the message flow. If the message flow fails, it is rolled back. If you choose **Automatic**, the ability to commit or roll back the action of the Database node on the database depends on the success or failure of the entire message flow.
 - **Commit**. If you want to commit any uncommitted actions performed in this message flow on the database connected to this node, irrespective of the success or failure of the message flow as a whole, select **Commit**. The changes to the database are committed even if the message flow itself fails.
4. Select **Basic** in the properties dialog navigator and set or clear the two check boxes:
 - If you want database warning messages to be treated as errors and the node to propagate the output message to the failure terminal, select the *Treat Warnings as Errors* check box. The box is initially cleared.

When you select the box, the node handles all positive return codes from the database as errors and generates exceptions in the same way as it does for the negative, or more serious, errors.

If you do not select the box, the node treats warnings as normal return codes, and does not raise any exceptions. The most significant warning raised is not found, which can be handled as a normal return code safely in most circumstances.

- If you want the broker to generate an exception when a database error is detected, select the *Throw Exception on Database Error* check box. The box is initially selected.

If you clear the box, include ESQL to check for any database error that might be returned after each database call that you make (you can use SQLCODE and SQLSTATE to do this). If an error has occurred, you must handle the error in the message flow to ensure the integrity of the broker and the database: the error is ignored if you do not handle it through your own processing, because you have chosen not to invoke the default error handling by the broker. For example, you can include the ESQL THROW statement to throw an exception in this node, or you can use the Throw node to generate your own exception at a later point.

5. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
6. Click **Apply** to make the changes to the Database node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The terminals of the Database node are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the input message is propagated if a failure is detected during the computation. If you have selected <i>Treat warnings as errors</i> , the node propagates the message to this terminal even if the processing completes successfully.
Out	The output terminal to which the transformed message is routed when processing in the node is completed. The transformed message might also be routed to this terminal by a PROPAGATE statement.
Out1	The first alternate output terminal to which the transformed message might be routed by a PROPAGATE statement.
Out2	The second alternate output terminal to which the transformed message might be routed by a PROPAGATE statement.
Out3	The third alternate output terminal to which the transformed message might be routed by a PROPAGATE statement.
Out4	The fourth alternate output terminal to which the transformed message might be routed by a PROPAGATE statement.

Note: See “PROPAGATE statement” on page 874 for the syntax of the PROPAGATE statement.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Database node Basic properties are described in the following table.

Property	M	C	Default	Description
Data Source	No	Yes		The ODBC data source name of the database in which reside the tables to which you refer in the ESQL associated with this node (identified by the <i>Statement</i> property).
Statement	Yes	No	Database	The name of the module within the ESQL file that contains the statements to execute against the database.
Transaction	Yes	No	Automatic	The transaction mode for the node. This can be Automatic or Commit.
Treat Warnings as Errors	Yes	No	Cleared	Treat database SQL warnings as errors. If you select the check box, this action is performed.
Throw Exception on Database Error	Yes	No	Selected	Database errors cause the broker to throw an exception. If you select the check box, this action is performed.

The Description properties of the Database node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

DataDelete node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 505
- “Configuring the DataDelete node” on page 505
- “Terminals and properties” on page 506

Purpose

Use the DataDelete node to interact with a database in the specified ODBC data source. The DataDelete node is a specialized form of the Database node, and the interaction is restricted to deleting one or more rows from a table within the database. You define what is deleted by defining mapping statements that use the data from the input message in some way to identify the action required.

You can set a property to control whether the update to the database is committed immediately, or deferred until the message flow completes, at which time the update is committed or rolled back according to the overall completion status of the message flow.

The DataDelete node is represented in the workbench by the following icon:



Using this node in a message flow

Consider a situation in which you are running a limited promotion. The goods are available only for the period of the promotion, and each customer can only have one item. When stocks of the sale goods run out, you want to remove their details from the stock database. When a message containing an order for the last item comes in, the DataDelete node is triggered to remove all the details for that item from the database.

Configuring the DataDelete node

When you have put an instance of the DataDelete node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the DataDelete node as follows:

1. Specify in *Data Source* the name by which the appropriate database is known on the system on which this message flow is to execute. The broker connects to this database with user ID and password information that you have specified on the `mqsicreatebroker`, `mqsichangebroker`, or `mqsisetdbparms` command. On z/OS systems, the broker uses the broker started task ID, or the user ID and password that were specified on the `mqsisetdbparms` command JCL, BIPSDBP in the customization data set `<hlq>.SBIPPROC`.

2. In *Statement*, identify the associated mapping routine that is to be executed in this node. By default, the name assigned to the mapping routine is identical to the name of the mappings file in which the routine is defined. The default name for the file is the name of the message flow concatenated with the name of the node when you include it in the message flow (for example, `MFlow1_DataDelete.mfmap` for the first DataDelete node in message flow `MFlow1`). You cannot specify a value that includes spaces.

If you click **Browse** next to this entry field, a dialog is displayed listing all available mapping routines that are accessible by this node. Select the routine that you want and click **OK**. The routine name is set in *Statement*.

To work with the mapping routine associated with this node, right-click the node and click **Open Mappings**. If the mapping routine does not exist, it is created for you with the default name in the default file. If the file already exists, you can also open file `<flow_name>_<node_name>.mfmap` in the Navigator view.

A mapping routine is specific to the type of node with which it is associated; you cannot use a mapping routine that you have developed for a DataDelete node with any other node that uses mappings (for example, a DataInsert node). If you create a mapping routine, you cannot call it from any other mapping routine, although you can call it from an ESQL routine.

3. Select the *Transaction* setting from the drop-down menu. The values are:
 - **Automatic** (the default). The message flow, of which the DataDelete node is a part, is committed if it is successful. That is, the actions that you define in the mappings are performed and the message continues through the message flow. If the message flow fails, it is rolled back. Therefore if you choose **Automatic**, the ability to commit or roll back the action of the DataDelete node on the database depends on the success or failure of the entire message flow.

- **Commit.** If you want to commit any uncommitted actions performed in this message flow on the database connected to this node, irrespective of the success or failure of the message flow as a whole, select **Commit**. The changes to the database are committed even if the message flow itself fails.
4. Select **Basic** in the properties dialog navigator and select or clear the two check boxes:
 - If you want database warning messages to be treated as errors, and the node to propagate the output message to the failure terminal, select the *Treat Warnings as Errors* check box. The box is initially cleared.
When you select the box, the node handles all positive return codes from the database as errors and generates exceptions in the same way as it does for the negative, or more serious, errors.
If you do not select the box, the node treats warnings as normal return codes, and does not raise any exceptions. The most significant warning raised is *not found*, which can be handled as a normal return code safely in most circumstances.
 - If you want the broker to generate an exception when a database error is detected, select the *Throw Exception on Database Error* check box. The box is initially selected.
If you clear the box, you must handle the error in the message flow to ensure the integrity of the broker and the database: the error is ignored if you do not handle it through your own processing, because you have chosen not to invoke the default error handling by the broker. For example, you could connect the failure terminal to an error processing subroutine.
 5. Select **Description** in the properties dialog navigator to enter a short description, a long description, or both.
 6. Click **Apply** to make the changes to the DataDelete node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The terminals of the DataDelete node are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the input message is propagated if a failure is detected during the computation. If you have selected <i>Treat warnings as errors</i> , the node propagates the message to this terminal even if the processing completes successfully.
Out	The output terminal that outputs the message following the execution of the database statement.

The following tables describe the node properties; the column headed **M** indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed **C** indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The DataDelete node Basic properties are described in the following table.

Property	M	C	Default	Description
Data Source	No	Yes		The ODBC data source name of the database in which reside the tables to which you refer in the mappings associated with this node (identified by the <i>Statement</i> property).
Statement	Yes	No	DataDelete	The name of the mapping routine that contains the statements that are to be executed against the database or the message tree. The routine is unique to this type of node.
Transaction	Yes	No	Automatic	The transaction mode for the node. This can be Automatic or Commit.
Treat Warnings as Errors	Yes	No	Cleared	Treat database SQL warnings as errors. If you select the check box, this action is performed.
Throw Exception on Database Error	Yes	No	Selected	Database errors cause the broker to throw an exception. If you select the check box, this action is performed.

The Description properties of the DataDelete node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

DataInsert node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 508
- “Configuring the DataInsert node” on page 508
- “Terminals and properties” on page 509

Purpose

Use the DataInsert node to interact with a database in the specified ODBC data source. The DataInsert node is a specialized form of the Database node, and the interaction is restricted to inserting one or more rows into a table within the database. You define what is inserted by defining mapping statements that use the data from the input message in some way to define the action required.

You can set a property to control whether the update to the database is committed immediately, or deferred until the message flow completes, at which time the update is committed, or rolled back according to the overall completion status of the message flow.

The DataInsert node is represented in the workbench by the following icon:



Using this node in a message flow

Consider a situation in which your company has developed a new product. The details about the product have been sent from your engineering department, and you need to extract details from the message and add them as a new row in your stock database.

Configuring the DataInsert node

When you have put an instance of the DataInsert node into a message flow you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the DataInsert node as follows:

1. Specify in *Data Source* the name by which the appropriate database is known on the system on which this message flow is to execute. The broker connects to this database with user ID and password information that you have specified on the `mqsicreatebroker`, `mqsichangebroker`, or `mqsisetdbparms` command. On z/OS systems, the broker uses the broker started task ID, or the user ID and password that were specified on the `mqsisetdbparms` command JCL, BIPSDBP in the customization data set `<hlq>.SBIPPROC`.
2. In *Statement*, identify the associated mapping routine that is to be executed in this node. By default, the name assigned to the mapping routine is identical to the name of the mappings file in which the routine is defined. The default name for the file is the name of the message flow concatenated with the name of the node when you include it in the message flow (for example, `MFlow1_DataInsert.mfmap` for the first DataInsert node in message flow `MFlow1`). You cannot specify a value that includes spaces.

If you click **Browse** next to this entry field, a dialog is displayed that lists all available mapping routines that are accessible by this node. Select the routine that you want and click **OK**. The routine name is set in *Statement*.

To work with the mapping routine associated with this node, right-click the node and select **Open Mappings**. If the mapping routine does not exist, it is created for you with the default name in the default file. If the file already exists, you can also open file `<flow_name>_<node_name>.mfmap` in the Navigator view.

A mapping routine is specific to the type of node with which it is associated; you cannot use a mapping routine that you have developed for a DataInsert node with any other node that uses mappings (for example, a DataDelete node). If you create a mapping routine, you cannot call it from any other mapping routine, although you can call it from an ESQL routine.

3. Select the *Transaction* setting from the drop-down menu. The values are:
 - **Automatic** (the default). The message flow, of which the DataInsert node is a part, is committed if it is successful. That is, the actions that you define in the mappings are taken and the message continues through the message flow. If the message flow fails, it is rolled back. Therefore if you choose **Automatic**, the ability to commit or roll back the action of the DataInsert node on the database depends on the success or failure of the entire message flow.
 - **Commit**. If you want to commit any uncommitted actions taken in this message flow on the database connected to this node, irrespective of the

success or failure of the message flow as a whole, select **Commit**. The changes to the database are committed even if the message flow itself fails.

4. Select **Basic** in the properties dialog navigator and set or clear the two check boxes:
 - If you want database warning messages to be treated as errors, and the node to propagate the output message to the failure terminal, select the *Treat Warnings as Errors* check box. The box is initially cleared.
 When you select the box, the node handles all positive return codes from the database as errors and generates exceptions in the same way as it does for the negative, or more serious, errors.
 If you do not select the box, the node treats warnings as normal return codes, and does not raise any exceptions. The most significant warning raised is not found, which can be handled as a normal return code safely in most circumstances.
 - If you want the broker to generate an exception when a database error is detected, select the *Throw Exception on Database Error* check box. The box is initially selected.
 If you clear the box, you must handle the error in the message flow to ensure the integrity of the broker and the database: the error is ignored if you do not handle it through your own processing, because you have chosen not to invoke the default error handling by the broker. For example, you could connect the failure terminal to an error processing subroutine.
5. Select **Description** in the properties dialog navigator to enter a short description, a long description, or both.
6. Click **Apply** to make the changes to the DataInsert node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
 Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The terminals of the DataInsert node are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the input message is propagated if a failure is detected during the computation. If you have selected <i>Treat warnings as errors</i> , the node propagates the message to this terminal even if the processing completes successfully.
Out	The output terminal that outputs the message following the execution of the database statement.

The following tables describe the node properties; the column headed **M** indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed **C** indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The DataInsert node Basic properties are described in the following table.

Property	M	C	Default	Description
Data Source	No	Yes		The ODBC data source name of the database in which reside the tables to which you refer in the mappings associated with this node (identified by the <i>Statement</i> property).
Statement	Yes	No	DataInsert	The name of the mapping routine that contains the statements that are to be executed against the database or the message tree. The routine is unique to this type of node.
Transaction	Yes	No	Automatic	The transaction mode for the node. This can be Automatic or Commit.
Treat Warnings as Errors	Yes	No	Cleared	Treat database SQL warnings as errors. If you select the check box, this action is performed.
Throw Exception on Database Error	Yes	No	Selected	Database errors cause the broker to throw an exception. If you select the check box, this action is performed.

The Description properties of the DataInsert node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

DataUpdate node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 511
- “Configuring the DataUpdate node” on page 511
- “Terminals and properties” on page 512

Purpose

Use the DataUpdate node to interact with a database in the specified ODBC data source. The DataUpdate node is a specialized form of the Database node, and the interaction is restricted to updating one or more rows from a table within the database. You define what is updated by defining mapping statements that use the data from the input message in some way to identify the action required.

You can set a property to control whether the update to the database is committed immediately, or deferred until the message flow completes, at which time the update is committed, or rolled back according to the overall completion status of the message flow.

The DataUpdate node is represented in the workbench by the following icon:



Using this node in a message flow

Consider a situation in which you have added the details of a new product, a keyboard, to your stock database. Now you have received a message from the Goods In department that indicates that 500 keyboards have been delivered to your premises. You can use the DataUpdate node to change the quantity of keyboards in your database from zero to 500.

Configuring the DataUpdate node

When you have put an instance of the DataUpdate node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the DataUpdate node as follows:

1. Specify in *Data Source* the name by which the appropriate database is known on the system on which this message flow is to execute. The broker connects to this database with user ID and password information that you have specified on the `mqsicreatebroker`, `mqsichangebroker`, or `mqsisetdbparms` command.

On z/OS systems, the broker uses the broker started task ID, or the user ID and password that were specified on the `mqsisetdbparms` command JCL, BIPsDBP in the customization data set <hlq>.SBIPPROC.

2. In *Statement*, identify the associated mapping routine that is to be executed in this node. By default, the name assigned to the mapping routine is identical to the name of the mappings file in which the routine is defined. The default name for the file is the name of the message flow concatenated with the name of the node when you include it in the message flow (for example, `MFlow1_DataUpdate.mfmap` for the first DataUpdate node in message flow `MFlow1`). You cannot specify a value that includes spaces.

If you click **Browse** next to this entry field, a dialog is displayed that lists all available mapping routines that are accessible by this node. Select the routine that you want and click **OK**. The routine name is set in *Statement*.

To work with the mapping routine associated with this node, right-click the node and click **Open Mappings**. If the mapping routine does not exist, it is created for you with the default name in the default file. If the file already exists, you can also open file <flow_name>_<node_name>.mfmap in the Navigator view.

A mapping routine is specific to the type of node with which it is associated; you cannot use a mapping routine that you have developed for a DataUpdate node with any other node that uses mappings (for example, a DataInsert node). If you create a mapping routine, you cannot call it from any other mapping routine, although you can call it from an ESQL routine.

3. Select the *Transaction* setting from the drop-down menu. The values are:
 - **Automatic** (the default). The message flow, of which the DataUpdate node is a part, is committed if it is successful. That is, the actions that you define in the mappings are performed and the message continues through the message flow. If the message flow fails, it is rolled back. Therefore if you choose **Automatic**, the ability to commit or roll back the action of the DataUpdate node on the database depends on the success or failure of the entire message flow.

- **Commit.** If you want to commit any uncommitted actions performed in this message flow on the database connected to this node, irrespective of the success or failure of the message flow as a whole, select **Commit**. The changes to the database are committed even if the message flow itself fails.
4. Select **Basic** in the properties dialog navigator and select or clear the two check boxes:
 - If you want database warning messages to be treated as errors, and the node to propagate the output message to the failure terminal, select the *Treat Warnings as Errors* check box. The box is initially cleared.
When you select the box, the node handles all positive return codes from the database as errors and generates exceptions in the same way as it does for the negative, or more serious, errors.
If you do not select the box, the node treats warnings as normal return codes, and does not raise any exceptions. The most significant warning raised is not found, which can be handled as a normal return code safely in most circumstances.
 - If you want the broker to generate an exception when a database error is detected, select the *Throw Exception on Database Error* check box. The box is initially selected.
If you clear the box, you must handle the error in the message flow to ensure the integrity of the broker and the database: the error is ignored if you do not handle it through your own processing, because you have chosen not to invoke the default error handling by the broker. For example, you could connect the failure terminal to an error processing subroutine.
 5. Select **Description** in the properties dialog navigator to enter a short description, a long description, or both.
 6. Click **Apply** to make the changes to the DataUpdate node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The terminals of the DataUpdate node are described in the following table:

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the input message is propagated if a failure is detected during the computation. If you have selected <i>Treat Warnings as Errors</i> , the node propagates the message to this terminal even if the processing completes successfully.
Out	The output terminal that outputs the message following the execution of the database statement.

The following tables describe the node properties; the column headed **M** indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed **C** indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The DataUpdate node Basic properties are described in the following table.

Property	M	C	Default	Description
Data Source	No	Yes		The ODBC data source name of the database in which reside the tables to which you refer in the mappings associated with this node (identified by the <i>Statement</i> property).
Statement	Yes	No	DataUpdate	The name of the mapping routine that contains the statements that are to be executed against the database or the message tree. The routine is unique to this type of node.
Transaction	Yes	No	Automatic	The transaction mode for the node. This can be Automatic or Commit.
Treat Warnings as Errors	Yes	No	Cleared	Treat database SQL warnings as errors. If you select the check box, this action is performed.
Throw Exception on Database Error	Yes	No	Selected	Database errors cause the broker to throw an exception. If you select the check box, this action is performed.

The Description properties of the DataUpdate node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Extract node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the Extract node” on page 514
- “Terminals and properties” on page 514

Purpose

Use the Extract node to extract the contents of the input message that you want to be processed by later nodes in the message flow. Using the Extract node, you can create a new output message that contains only a subset of the contents of the input message. The output message comprises only those elements of the input message that you specify for inclusion when configuring the Extract node, by defining mapping statements.

The Extract node is represented in the workbench by the following icon:



Using this node in a message flow

You might find this node useful if you require only a subset of the message after initial processing of the whole message. For example, you might want to store the whole message for audit purposes (in the Warehouse node), but propagate only a small part of the message (order information, perhaps) for further processing.

You receive orders from new clients and you want to collect their names and addresses for future promotions. To do this, you use the Extract node to get this information from each order, and send it as a new message to head office. These messages are processed at head office so that the customer details can be included in the next marketing campaign.

Configuring the Extract node

When you have put an instance of the Extract node into a message flow you can configure it. Right-click the node in the editor view and select **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Extract node as follows:

1. In *Mapping Module*, identify the associated mapping routine that is to be executed in this node. By default, the name assigned to the mapping routine is identical to the name of the mappings file in which the routine is defined. The default name for the file is the name of the message flow concatenated with the name of the node when you include it in the message flow (for example, MFlow1_Extract.mfmap for the first Extract node in message flow MFlow1). You cannot specify a value that includes spaces.

If you click **Browse** next to this entry field, a dialog is displayed that lists all available mapping routines that are accessible by this node. Select the routine that you want and click **OK**. The routine name is set in *Mapping Module*.

To work with the mapping routine associated with this node, right-click the node and select **Open Mappings**. If the mapping routine does not exist, it is created for you with the default name in the default file. If the file already exists, you can also open file <flow_name>_<node_name>.mfmap in the Navigator view.

A mapping routine is specific to the type of node with which it is associated; you cannot use a mapping routine that you have developed for an Extract node with any other node that uses mappings (for example, a DataInsert node). If you create a mapping routine, you cannot call it from any other mapping routine, although you can call it from an ESQL routine.

2. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
3. Click **Apply** to make the changes to the Extract node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Extract node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the input message is routed if a failure is detected during extraction.
Out	The output terminal to which the transformed message is routed if the input message is processed successfully.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Extract node Basic properties are described in the following table.

Property	M	C	Default	Description
Mapping Module	Yes	No	Extract	The name of the mapping routine that contains the statements to execute against the message tree. The routine is unique to this type of node.

The Extract node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Filter node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 516
- “Configuring the Filter node” on page 516
- “Terminals and properties” on page 518

Purpose

Use the Filter node to route a message according to message content. You define the route by coding a filter expression in ESQL. You can include elements of the input message or message properties in the filter expression. You can also use data held in an external database to complete the expression. The output terminal to which the message is routed depends on whether the expression evaluates to true, false, or unknown.

Connect the terminals that cover all situations that could result from the filter; if the node propagates the message to a terminal that is not connected, the message is discarded even if it is transactional.

This node accepts ESQL statements in the same way as the Compute and Database nodes. The last statement executed must be a RETURN <expression> statement whose expression evaluates to a Boolean value. This Boolean value determines which terminal the message is routed to. In many cases, the routing algorithm is a simple comparison of message field values. The comparison is described by the expression and the RETURN statement is the only statement. If you code RETURN without an expression (RETURN;) or with a NULL expression, the node propagates the message to the unknown terminal.

If your message flow requires more complex routing options, you can use the RouteToLabel and Label nodes.

The Filter node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following samples for examples of how you can use this node:

- Airline Reservations sample
- Scribble sample
- Error Handler sample
- Large Messaging sample

Consider a situation in which you have produced an online test with 10 multiple choice questions. Each message coming in has a candidate name and address followed by a series of answers. Each answer is checked, and if it is correct the field SCORE is incremented by 1. When all the answers have been checked, the field SCORE is tested to see if it is greater than 5. If it is, the Filter node propagates the message to the flow which handles successful candidate input; otherwise the message is filtered into the rejection process, and a rejection message is created.

Configuring the Filter node

When you have put an instance of the Filter node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Filter node as follows:

1. Specify in *Data Source* the name by which the appropriate database is known on the system on which this message flow is to execute. The broker connects to this database with user ID and password information that you have specified on the `mqsicreatebroker`, `mqsichangebroker`, or `mqsisetdbparms` command.

On z/OS systems, the broker uses the broker started task ID, or the user ID and password that were specified on the `mqsisetdbparms` command JCL, `BIPSDBP` in the customization data set `<hlq>.SBIPPROC`.

2. In *Filter Expression*, identify the module within an ESQL file that contains the ESQL statements the node executes. If you want the module name to include one or more spaces, enclose it in double quotes in the *Filter Expression* property. Code ESQL statements to customize the behavior of the Filter node in an ESQL file that is associated with the message flow in which you have included this instance of the Filter node. The ESQL file, which by default has the name `<message_flow_name>.esql`, contains ESQL for every node in the message flow that requires it. Each portion of code that is related to a specific node is known as a module.

If an ESQL file does not already exist for this message flow, right-click the Filter node and click **Open ESQL**. This creates and opens a new ESQL file in the ESQL editor view.

If the file already exists, click the **Browse** button beside the *Filter Expression* property. This displays the Module Selection dialog, which lists the available Filter node modules defined in the ESQL files that are accessible by this

message flow (ESQL files can be defined in other, dependent, projects). Select the appropriate module and click **OK**. If no suitable modules are available, the list is empty.

If the module you specify doesn't exist, that module is created for you, and the editor positions the file to display it. If the file and the module already exist, the editor positions the file and displays and highlights the correct module.

If you prefer, you can open the appropriate ESQL file in the Resource Navigator and select this node in the Outline view.

If a module skeleton is created for this node in a new or existing ESQL file, it consists of the following ESQL. The default module name is shown in this example:

```
CREATE FILTER MODULE <flow_name> Filter
    CREATE FUNCTION Main() RETURNS BOOLEAN
    BEGIN
        RETURN TRUE;
    END;
END MODULE;
```

If you create your own ESQL module, you must create exactly this skeleton. You can update the default name, but ensure that the name that you specify matches the name of the corresponding node property *Filter Expression*.

Add your own ESQL to customize this node after the **BEGIN** statement, and before the **RETURN** statement. If the expression on the **RETURN** statement is not **TRUE** or **FALSE**, its value is resolved to determine the terminal to which the message is propagated. If the expression resolves to **NULL**, or you code **RETURN;**, or you omit the **RETURN** statement, the node propagates the message to the unknown terminal.

You can also click **File** → **New** → **Message Flow ESQL File** to create an ESQL file.

You can use all the ESQL statements including **SET**, **WHILE**, **DECLARE**, and **IF** in this module, but (unlike the Compute node) the Filter node propagates the message that it receives at its input terminal to its output terminal unchanged. This means that, like the Database node, you have only one message to refer to in a Filter node.

Because you can't modify any part of any message, the assignment statement (the **SET** statement, not the **SET** clause of the **INSERT** statement) can assign values only to temporary variables. The scope of actions that you can take with an assignment statement is therefore limited.

3. Select the *Transaction* setting from the drop-down menu. The values are:
 - **Automatic** (the default). The message flow, of which the Filter node is a part, is committed if it is successful. That is, the actions that you define in the ESQL module are performed and the message continues through the message flow. If the message flow fails, it is rolled back. Therefore if you choose **Automatic**, the ability to commit or rollback the action of the Filter node on the database depends on the success or failure of the entire message flow.
 - **Commit**. If you want to commit any uncommitted actions performed in this message flow on the database connected to this node, irrespective of the success or failure of the message flow as a whole, select **Commit**. The changes to the database are committed even if the message flow itself fails.
4. Select **Basic** in the properties dialog navigator and set or clear the two check boxes:
 - If you want database warning messages to be treated as errors and propagate the output message from the node to the failure terminal, select the *Treat warnings as errors* check box. The box is initially cleared.

When you select the box, the node handles all positive return codes from the database as errors and generates exceptions in the same way as it does for the negative, or more serious, errors.

If you do not select the box, the node treats warnings as normal return codes, and does not raise any exceptions. The most significant warning raised is not found, which can be handled as a normal return code safely in most circumstances.

- If you want the broker to generate an exception when a database error is detected, select the *Throw exception on database error* check box. The box is initially selected.

If you clear the box, you must include ESQL to check for any database error that might be returned after each database call you make (you can use SQLCODE and SQLSTATE to do this). If an error has occurred, you must handle the error in the message flow to ensure the integrity of the broker and the database: the error is ignored if you do not handle it through your own processing, because you have chosen not to invoke the default error handling by the broker. For example, you can include the ESQL THROW statement to throw an exception in this node, or you can use the Throw node to generate your own exception at a later point.

5. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
6. Click **Apply** to make the changes to the Filter node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Filter node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if a failure is detected during the computation.
Unknown	The output terminal to which the message is routed if the specified filter expression evaluates to unknown or null.
False	The output terminal to which the message is routed if the specified filter expression evaluates to false.
True	The output terminal to which the message is routed if the specified filter expression evaluates to true.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Filter node Basic properties are described in the following table.

Property	M	C	Default	Description
Data Source	No	Yes		The ODBC data source name of the database in which reside the tables to which you refer in the ESQL associated with this node (identified by the <i>Filter Expression</i> property).
Transaction	Yes	No	Automatic	The transaction mode for the node. This can be Automatic or Commit.
Filter Expression	Yes	No	Filter	The name of the module within the ESQL resource (file) that contains the statements to execute against the message that is received in the node.
Treat warnings as errors	Yes	No	Cleared	Treat database SQL warnings as errors. If you select the check box, this action is performed.
Throw exception on database error	Yes	No	Selected	Database errors cause the broker to throw an exception. If you select the check box, this action is performed.

The Filter node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

FlowOrder node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 520
- “Configuring the FlowOrder node” on page 520
- “Terminals and properties” on page 521

Purpose

Use the FlowOrder node to control the order in which a message is processed by a message flow. The input message is propagated to the first output terminal and the sequence of nodes connected to this terminal process the message. When that message processing is complete, control returns to the FlowOrder node. If the message processing completes successfully, the input message is propagated to the second output terminal and the sequence of nodes connected to this terminal processes the message.

The message that is propagated through the second output terminal is the input message; it is not modified in any way, even if the sequence of nodes connected to first terminal has modified the message.

You can include this node in a message flow at any point where the order of execution of subsequent nodes is important.

If you connect multiple nodes to the first output terminal, the second output terminal, or both, the order in which the multiple connections on each terminal are processed is random and unpredictable. However, the message is propagated to all

target nodes connected to the first output terminal, which must all complete successfully, before it is propagated to any node connected to the second output terminal.

Your message flow performance can benefit from including the FlowOrder node in a situation where one sequence of processing required for a message is significantly shorter than another sequence of processing. If you connect the shorter sequence to the first terminal, any failure is identified quickly and prevents execution of the second longer sequence of processing.

The FlowOrder node is represented in the workbench by the following icon:



Using this node in a message flow

For an example of using this node, assume that your company receives orders from customers using the Internet. When the order is received, it is processed by nodes connected to the first terminal of a FlowOrder node to debit the stock level in your database and raise an invoice. A check is made to see whether the customer has indicated that his details can be sent to other suppliers. If the customer has indicated that he does not want this information to be divulged, this check fails and no further processing occurs. If the customer is happy for you to share his details with other companies (that is, the test is successful), the input message is propagated to the second terminal so that the customer's details can be added to the mailing list.

Configuring the FlowOrder node

When you have put an instance of the FlowOrder node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the FlowOrder node as follows:

1. Select **Description** in the properties dialog navigator to enter a short description, a long description, or both.
2. Click **Apply** to make the changes to the FlowOrder node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

The FlowOrder node has no configurable properties that impact its operation. You determine how it operates by connecting the first and second output terminals to subsequent nodes in your message flow.

1. Connect the first terminal to the first node in the sequence of nodes that provide the first phase of processing this message. This can be a sequence of one or more nodes that perform any valid processing. It can conclude with an output node, but does not have to.
2. Connect the second terminal to the first node in the sequence of nodes that provide the second phase of processing this message. This can be a sequence of one or more nodes that perform any valid processing. It can conclude with an output node, but does not have to.

The message that is propagated through the second terminal is identical to that propagated through the first terminal. Any changes that you have introduced as a result of the first phase of processing are ignored by this node.

If the first phase of processing fails, the FlowOrder node does not regain control and does not propagate the message through the second terminal.

Terminals and properties

The FlowOrder node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if a failure is detected during the computation.
First	The output terminal to which the input message is routed in the first instance.
Second	The output terminal to which the input message is routed in the second instance. The message is routed to this terminal only if routing to First is successful.

The following table describes the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The FlowOrder node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

HTTPInput node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 523
- “Configuring the HTTPInput node” on page 523
- “Terminals and properties” on page 525

Purpose

Use the HTTPInput node to receive Web service requests for processing by a message flow. Using the HTTPInput node with the HTTPReply and HTTPRequest nodes, the broker can act as an intermediary for Web services, and Web service

requests can be transformed and routed in the same way as other message formats supported by WebSphere Message Broker. Web Service requests can be received either in standard HTTP (1.0 or 1.1) format, and also in HTTP over SSL (HTTPS) format. You can set the *Use HTTPS* property to choose whether to handle HTTP or HTTPS requests.

If you include an HTTPInput node in a message flow, you must either include an HTTPReply node in the same flow, or pass the message to another flow that includes an HTTPReply node (for example, through an MQOutput node to a second flow that starts with an MQInput node). In the latter case, the request from, and reply to, the client are coordinated by the request identifier stored in the LocalEnvironment (described below).

The HTTPInput node handles messages in the following message domains:

- MRM
- XML
- XMLNS
- XMLNSC
- JMS
- JMSStream
- IDOC
- MIME
- BLOB

When the HTTPInput node receives a message from a Web service client, it invokes the appropriate parsers to interpret the headers and the body of the message, and to create the message tree that is used internally by the message flow. The node creates a unique identifier for the input message and stores it as a binary array of 24 bytes in the LocalEnvironment tree at `LocalEnvironment.Destination.HTTP.RequestIdentifier`. This value is used by the HTTPReply node and must not be modified in any way.

HTTP messages are always non-persistent, and have no associated order.

HTTP messages are non-transactional. However, if the message flow interacts with a database or another external resource such as a WebSphere MQ queue, these interactions are performed transactionally. The HTTPInput node provides commit or rollback depending on how the message flow has ended, and how it is configured for error handling (how failure terminals are connected, for example). If the message flow is rolled back by this node, a fault message is generated and returned to the client. The format of the fault is defined by the Fault Format property

If an exception occurs downstream in this message flow, and is not caught but is returned to this node, the node constructs an error reply to the client. This error is derived from the exception and the format of the error is defined by the Fault Format property.

If you include an output node in a message flow that starts with an HTTPInput node, it can be any of the supported output nodes (including user-defined output nodes). You can create a message flow that receives messages from Web service clients and generates messages for clients that use all supported transports to connect to the broker, because you can configure the message flow to request the broker to provide any conversion that is necessary.

If you create a message flow to use as a subflow, you cannot use a standard input node; you must use an instance of the Input node as the first node to create an in terminal for the subflow.

If your message flow does not receive Web service requests, you can choose one of the supported input nodes.

The HTTPInput node is represented in the workbench by the following icon:



Using this node in a message flow

The HTTPInput node can be used in any message flow that needs to accept HTTP or HTTPS messages. The most common example of this is a message flow that implements a web service. For information on web service applications, see *Web service applications*.

Configuring the HTTPInput node

When you have put an instance of the HTTPInput node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed in the properties dialog.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the HTTPInput node as follows:

1. In *URL Selector*, put the path part of the URL from which this node receives Web service requests. Do not give the full URL.
2. Enter the *Maximum client wait time* timeout interval, as a number of seconds. This is the length of time that the TCP/IP listener that received the input message from the Web service client waits for a response from the HTTPReply node in the message flow. If a response is received within this time, the listener propagates the response to the client. If a response is not received in this time, the listener sends a SOAP Fault message to the client that indicates that its timeout has expired.
3. Select the *Fault Format* as one of SOAP 1.1, SOAP 1.2 or HTML.
4. If the node is to accept secure HTTP, select the *Use HTTPS* check box.
5. Select Default in the properties dialog navigator and set values for the properties describing the message domain, message set, message type, and message format that the node uses to determine how to parse the incoming message.
 - In the *Message Domain* field, select the name of the parser that you are using from the drop-down list. You can choose from:
 - MRM
 - XML
 - XMLNS
 - XMLNSC
 - JMSMap
 - JMSStream
 - IDOC
 - MIME

- BLOB
 - If you are using the MRM or IDOC parser, select the correct message set from the drop-down list in the *Message Set* field. This list is populated with available message sets when you select MRM or IDOC as the domain.
Leave the *Message Set* field blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
 - If you are using the MRM parser, select the correct message from the drop-down list in *Message Type*. This list is populated with messages that are defined in the message set that you have selected.
Leave *Message Type* blank for XML, XMLNS, XMLNSC, JMS, IDOC, MIME, and BLOB parsers.
 - If you are using the MRM or IDOC parser, Select the format of the message from the drop-down list in the *Message Format* field. This list includes all the physical formats that you have defined for this message set. da
Leave the *Message Format* field blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
6. Select Validation in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)
For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.
 7. Select General Message Options in the properties dialog navigator. *Parse Timing* is, by default, set to On Demand. This causes validation to be delayed until it is parsed by partial parsing. If you change this to Immediate, partial parsing is overridden and everything in the message is parsed and validated, except those complex types with a Composition of Choice or Message that cannot be resolved at the time. If you change this to Complete, partial parsing is overridden and everything in the message is parsed and validated; complex types with a Composition of Choice or Message that cannot be resolved at the time cause a validation failure.
 8. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
 9. Click **Apply** to make the changes to the HTTPInput node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

HTTPInput routes each message that it retrieves successfully to the out terminal. If message validation fails, the message is routed to the failure terminal; you can connect nodes to this terminal to handle this condition. If you have not connected the failure terminal, the message is discarded, the *Maximum client wait time* expires, and the TCP/IP listener returns an error to the client. There are no other situations in which the message is routed to the failure terminal.

If the message is caught by this node after an exception has been thrown further on in the message flow, the message is routed to the catch terminal. If you have not connected the catch terminal, the message is discarded, the *Maximum client wait time* expires, and the TCP/IP listener returns an error to the client.

Terminals and properties

The HTTPInput node terminals are described in the following table.

Terminal	Description
Failure	The output terminal to which the message is routed if an error occurs.
Out	The output terminal to which the message is routed if it is successfully retrieved.
Catch	The output terminal to which the message is routed if an exception is thrown downstream and caught by this node.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The HTTPInput node Basic properties are described in the following table.

Property	M	C	Default	Description
URL Selector	Yes	Yes		Identifies the location from where Web service requests are retrieved. If the URL that you want is <code>http://<hostname>[:<port>]/[<path>]</code> , specify either <code>/<path></code> or <code>/<path fragment>/*</code> where <code>*</code> is a wild card that you can use to mean match any.
Maximum Client Wait Time	Yes	No	180	How long the listener waits, in seconds, before sending an error message back to the client. The valid range is zero (which means an indefinite wait) to $(2^{31})-1$.
Fault Format	No	Yes	SOAP 1.1	The property value can be SOAP 1.1, SOAP 1.2 or HTML. This property defines the format of any HTTP errors that are returned to the client.
Use HTTPS	No	Yes	no	Identifies whether the node is to accept secure HTTP.

The HTTPInput node Default properties are described in the following table.

Property	M	C	Default	Description
Message Domain	No	No		The domain that will be used to parse the incoming message.
Message Set	No	No		The name or identifier of the message set in which the incoming message is defined.
Message Type	No	No		The name of the incoming message.
Message Format	No	No		The name of the physical format of the incoming message.

The Validation properties of the HTTPInput node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	No	Yes	None	Whether validation takes place. Valid values are None, Content and Value, and Content.
Failure Action	No	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	No	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	No	No	None	This property cannot be edited.

The properties of the General Message Options for the HTTPInput node are described in the following table.

Property	M	C	Default	Description
Parse Timing	No	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to “Parsing on demand” on page 706 for a full description of this property.
Use MQRFH2C Compact Parser for MQRFH2 Domain	No	No	Cleared	This property controls whether the MQRFH2C Compact Parser, instead of the MQRFH2 parser, is used for MQRFH2 headers.

The properties of the XMLNSC Parser Options for the HTTPInput node are described in the following table.

Property	M	C	Default	Description
Use XMLNSC Compact Parser for XMLNS Domain	No	No	Cleared	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.
Mixed Content Retain Mode	No	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.
Comments Retain Mode	No	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.
Processing Instructions Retain Mode	No	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The HTTPInput node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

HTTPReply node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the HTTPReply node”
- “Terminals and properties” on page 528

Purpose

Use the HTTPReply node to return a response from the message flow to the Web service client. This node generates the response to the Web service client from which the input message was received by the HTTPInput node, and waits for confirmation that it has been sent.

If you include an HTTPReply node in a message flow, you must either include an HTTPInput node in the same flow, or the message must be received from another flow that started with an HTTPInput node. The response is associated with the reply by a request identifier that is stored in LocalEnvironment by the HTTPInput node.

This node constructs a reply message for the Web service client from the entire input message tree, and returns it to the requestor.

The HTTPReply node is represented in the workbench by the following icon:



Using this node in a message flow

Refer to the HTTPInput node for examples of how you can use this node.

Configuring the HTTPReply node

When you have put an instance of the HTTPReply node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node’s basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the HTTPReply node as follows:

1. Select the *Ignore Transport Failures* check box if you want transport-related failures to be ignored (for example, if the client is disconnected). If you clear

the check box, and a transport-related error occurs, the input message is propagated to the failure terminal. If you clear the check box, you must supply a value for *Reply send timeout*.

2. Set the *Reply send timeout* value if you are not ignoring transport failures. This is the length of time that the node waits for an acknowledgment that the client has received the reply. If the acknowledgment is received within this time, the input message is propagated through the out terminal to the rest of the message flow, if it is connected. If an acknowledgment is not received within this time, the input message is propagated through the failure terminal, if it is connected. If the failure terminal is not connected, and an acknowledgment is not received in time, an exception is generated.
3. Make sure that the *Generate default HTTP headers from reply or response* check box is selected if you want the default Web service headers to be created using values from the HTTPReplyHeader or the HTTPResponseHeader. If the appropriate header is not present in the input message, default values are used. The node always includes a Content-Length header, set to the correct calculated value, in the HTTPReplyHeader, even if this was not included in the original request.
4. Select Validation in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)
For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.
5. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
6. Click **Apply** to make the changes to the HTTPReply node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the output terminals to another node:

Connect the out or failure terminal of this node to another node in this message flow if you want to process the message further, process errors, or send the message to an additional destination.

Terminals and properties

The HTTPReply node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if a failure is detected when the message is propagated.
Out	The output terminal to which the message is routed if it has been successfully propagated, and if further processing is required within this message flow.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C

indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The HTTPReply node Basic properties are described in the following table.

Property	M	C	Default	Description
Ignore Transport Failures	Yes	No	Selected	Whether transport-related failures are ignored. If you select the check box, this action is performed.
Reply send timeout	Yes	No	120	The time in seconds that the reply node waits before assuming the reply has failed to reach the client. The valid range is zero (which means an indefinite wait) to $(2^{31})-1$. Valid only if <i>Ignore Transport Failures</i> is cleared.
Generate default HTTP headers from reply or response	Yes	No	Selected	The check box is selected if the default Web service headers are created using values from the HTTPReplyHeader or HTTPResponseHeader.

The Validation properties of the HTTPReply node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	Inherit	Whether validation takes place. Valid values are None, Content and Value, Content, and Inherit.
Failure Action	Yes	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	Yes	No	None	This property cannot be edited.

The HTTPReply node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

HTTPRequest node

This topic contains the following sections:

- “Purpose” on page 530
- “Using this node in a message flow” on page 530
- “Configuring the HTTPRequest node” on page 532
- “Terminals and properties” on page 537

Purpose

Use the HTTPRequest node to interact with a Web service, using all or part of the input message as the request sent to that service. You can also configure the node to create a new output message from the contents of the input message augmented by the contents of the Web service response before you propagate the message to subsequent nodes in the message flow.

Depending on the configuration, this node constructs an HTTP or an HTTP over SSL (HTTPS) request from the specified contents of the input message and sends this to the Web service. It receives the response from the Web service, and parses the response for inclusion in the output tree. It generates HTTP headers if these are required by your configuration.

You can use this node in a message flow that does not contain an HTTPInput or HTTPReply node.

The HTTPRequest node handles messages in the following message domains:

- MRM
- XML
- XMLNS
- XMLNSC
- JMSMap
- JMSStream
- MIME
- BLOB
- IDOC

The HTTPRequest node is represented in the workbench by the following icon:



Using this node in a message flow

The HTTPRequest node can be used in any message flow that needs to send an HTTP request. The most common example of this is a message flow that invokes a Web service. For information on Web service applications, see [Web service applications](#).

Handling errors

Because the node interacts directly with an external service using TCP/IP, it can experience the following types of error:

1. Errors that are generated by TCP/IP, for example no route to host or connection refused.

If the node detects these errors, it generates an exception, populates the exception list with the error information received, and routes the input message unchanged to the failure terminal.

2. Errors that are returned by the Web server. These are HTTP status codes outside the range 100 through 299. If the node detects these errors, it routes the reply to the node's error terminal whilst following the properties specified on the Error tab on the request node.

The reply will be output as a BLOB, as the node cannot determine what format the reply will be. If you have not configured this node to handle redirection, messages with a redirection status code (3xx) are also handled in this way.

HTTP Response Codes

The HTTPRequest node treats the 100 series status codes as a continue response, discards the current response and waits for another response from the Web server.

The 200 series status codes are treated as success and the response is routed to the out terminal of the node, whilst following the settings on the various tabs on the node for the format of the output message generated.

The 300 series status codes are for redirection. If the *Follow Redirection* property is selected, the node does not resend the request to the new specified in the response received. If the *Follow Redirection* property is not selected, the codes is treated as an error as described in the section above on handling errors.

The 400 and 500 series status codes are errors, and are treated as described section above on handling errors.

Manipulating headers

If the *Replace input message with web-service response* property, or *Replace input with error* property is selected, the header for the input message, that is, the header that belongs to the message as it arrives at the IN terminal of the HTTPRequest node, is not propagated with the message that leaves the HTTPRequest node. However, if one of the properties that specifies a location in the message tree is specified, the input message's headers are propagated.

The HTTPResponse header, which contains the headers that are returned by the remote Web service, is the first header in the message that is propagated from the node, after properties. This is true regardless of the options that are chosen. Therefore, if you want the reply from the HTTPRequest node to be put to an MQ queue, you must manipulate the headers so that an MQMD is the first header (after properties).

If you are replacing the input message with a response, you can copy the input message's MQMD to the Environment tree before the HTTPRequest node, and then copy it back into the message tree after the HTTPRequest node. If you are specifying a location for the response, in order to maintain existing input message headers, you must move or remove the HTTP Response header so that the MQMD is the first header.

The following is an example containing ESQL that removes the HTTPHeader:

```
SET OutputRoot = InputRoot;  
SET OutputRoot.HTTPResponseHeader = NULL;
```

The following is an example containing ESQL for moving the HTTPHeader, and therefore preserving the information that it provides:

```
SET OutputRoot = InputRoot;  
DECLARE HTTPHeaderRef REFERENCE TO OutputRoot.HTTPResponseHeader;  
DETACH HTTPHeaderRef;  
ATTACH HTTPHeaderRef TO OutputRoot.MQMD AS NEXTSIBLING;
```

Enabling HTTP GET

By default the HTTPRequest node uses the HTTP POST method when connecting to the remote Web server. To enable the request node to use the HTTP GET method instead, you must set a field in the output local environment tree, as shown in the following example:

```
SET OutputLocalEnvironment.Destination.HTTP.RequestLine.Method = 'GET';
```

Configuring the HTTPRequest node

When you have put an instance of the HTTPRequest node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the HTTPRequest node as follows:

1. The HTTPRequest node determines the URL for the Web service to which it sends a request. You must set one of the following three options; the node checks these in the order shown (that is, the first always overrides the second, the second overrides the third):
 - a. X-Original-HTTP-URL in the HTTPRequest header in the input message
 - b. LocalEnvironment.Destination.HTTP.RequestURL in the input message
 - c. The *Web Service URL* property

The first two options provide you with dynamic methods to set a URL for each input message as it passes through the message flow. If you want to use either of these options, you must include a Compute node in the message flow before the HTTPRequest node to create and initialize the required value.

The third option provides a value that is fixed for every message received in this node. You must set this property to contain a default setting that is used if the other fields have not been created, or contain a null value. If either field contains a value, the setting of this property is ignored. The *Web Service URL* property must contain a valid URL or the deploy will fail. You must also ensure that the value that you set in X-Original-HTTP-URL or the LocalEnvironment.Destination.HTTP.RequestURL is also a valid URL; if it is not, the node generates an exception and the message is propagated to the failure terminal.

If a URL begins `http://` the request node makes an HTTP request to the specified URL. If the URL begins `https://` the request node makes an HTTP over SSL (HTTPS) request to the specified URL, using the parameters that are specified in the SSL tab on the node.

2. Set the *Request timeout* value. This is the length of time that the node waits for a response from the Web service. If a response is received within this time, the reply is propagated through the out terminal to the rest of the message flow. If a response is not received within this time, the input message is propagated through the failure terminal, if it is connected. If the failure terminal is not connected, and a response is not received in this time, an exception is generated.
3. In *HTTP(S) Proxy Location*, set the location of the proxy server to which requests are sent.
4. Select or clear the *Follow HTTP(S) redirection* check box to specify how the node handles Web service responses that contain an HTTP status code of 300 through 399:

- If you select the check box, the node follows the redirection provided in the response and reissues the Web service request to the new URL included in the message content.
 - If you clear the check box, the node does not follow the redirection provided. The response message is propagated to the error terminal.
5. Select the *HTTP Version*. The possible values for the HTTP Version property are 1.0 and 1.1.
- If you select HTTP/1.1, you have the option of also using HTTP/1.1 keep-alive.
6. If you are planning on using HTTP over SSL (HTTPS) requests, select the SSL tab and set the values for https requests.
- Specify the protocol that is to be used to make the request. The following options are available:
 - *SSL* (the default). This tries to connect using the SSLv3 protocol first, but allows the handshake fall back to the SSLv2 protocol where the SSLv2 protocol is supported by the underlying JSSE provider.
 - *SSLv3*. This tries to connect with the SSLv3 protocol only. Fall back to SSLv2 is not allowed.
 - *TLS*. This tries to connect with the TLS protocol only. Fall back to SSLv3 or SSLv2 is not allowed.

Note that both ends of an SSL connection must agree on the protocol to use, so the chosen protocol must be one that the remote server can accept.
 - Specify the *Allowed SSL Ciphers*. This setting allows you to specify a single cipher (such as `SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA`) or a list of ciphers that will be the only ones used by the connection. This set of ciphers must include one or more that are accepted by the remote server. A comma is used as a separator between the ciphers. The default (an empty string) allows the node to use any, or all or the available ciphers during the SSL connection handshake, which allows the greatest scope for making a successful SSL connection.
7. Select Advanced in the properties dialog navigator and set values for the Advanced properties that describe the structure and content of the Web service request and response.
- a. Specify the content of the request message that is sent to the Web service:
- If you want the request message to be the whole input message body, leave the *Use whole input message as request* check box selected (this is its default setting).

If you want the request message to contain a subset of the input message, clear this check box and complete the property *Request message location in tree*.
 - In *Request message location in tree*, enter the start location from which the content of the input message tree is copied to the request message. This property is required only if you have cleared the *Use whole input message as request* property. The node creates a new request message and copies the specified parts of the input message (the input message itself is not modified).

You can enter any valid ESQL field reference, including expressions within the reference. For example, enter:

```
InputRoot.XMLNS.ABC
```

If you select the *Use whole input message as request* check box, this property is ignored.

When the appropriate message tree content is parsed to create a bit stream, the message properties (domain, set, type, and format) associated with the input message body and stored in the Properties folder are used.

- b. Specify the content of the output message that is propagated to the next node in the message flow:
- If you want the whole Web service response message to be propagated as the output message, leave the *Replace input message with web-service response* check box selected (this is its default setting).
If you want the Web service response message to be included in the output message with part of the input message content, clear this check box and complete the property *Response message location in tree*. If you clear this property, the node copies the input message to the output message and writes the Web service response message over the output message content at the specified location (the input message itself is not modified).
 - In *Response message location in tree*, enter the start location within the output message tree at which the parsed elements from the Web service response message bit stream are stored. This property is required only if you have cleared the *Replace input message with web-service response* property.

You can enter any valid ESQL field reference, including expressions within the reference, and including new field references (to create a new node in the message tree for the response). For example, enter:

```
OutputRoot.XMLNS.ABC.DEF
```

or

```
Environment.WSReply
```

If you select the *Replace input message with web-service response* check box, this property is ignored.

When the response bit stream is parsed to create message tree contents, the message properties (domain, set, type, and format) that you have specified in the node Default properties (described below) are used.

- c. If you want the node to generate an HTTPRequestHeader for the request message, leave the *Generate default HTTP headers from input* check box selected (this is the default setting).

If you do not want the node to generate an HTTPRequestHeader for the request message, clear the *Generate default HTTP headers from input* check box (the default setting is selected). To control the contents of the HTTPRequestHeader that is included in the request message, include a Compute node that adds an HTTPRequestHeader to the input message before this HTTPRequest node in the message flow, and clear this check box.

- If you have selected *Generate default HTTP headers from input* and the input message includes an HTTPRequestHeader, the node extracts Web service headers from the input HTTPRequestHeader and adds any unique Web service headers, except Host (see table below), that are present in an HTTPInputHeader, if one exists in the input message. (An HTTPInputHeader might be present if the input message has been received from a Web service by the HTTPInput node.)

It also adds the Web service headers shown in the following table, with default values, if these are not present in the HTTPRequestHeader or the HTTPInputHeader.

Header	Default value
SOAPAction	"" (empty string)
Content-Type	text/xml; charset=utf-8
Host	The hostname to which the request is to be sent

It also adds the optional header Content-Length with the correct calculated value, even if this is not present in the HTTPRequestHeader or the HTTPInputHeader.

- If you have selected *Generate default HTTP headers from input* and the input message does not include an HTTPRequestHeader, the node extracts Web service headers, except Host, from the HTTPInputHeader, if it is present in the input message. It adds the required Web service headers with default values, if these are not present in the HTTPInputHeader.
- If you have cleared *Generate default HTTP headers from input* and the input message includes an HTTPRequestHeader, the node extracts all Web service headers present in the input HTTPRequestHeader. It does not check for the presence of an HTTPInputHeader in the input message, and it does not add the required Web service headers if they are not supplied by the input HTTPRequestHeader.
- If you have cleared *Generate default HTTP headers from input* and the input message does not include an HTTPRequestHeader, no Web service headers are generated. The node does not check for the presence of an HTTPInputHeader in the input message and does not add any required Web service header. The request message is propagated to the Web service without an HTTPRequestHeader. This typically causes an error to be generated by the Web service, unless the Web service is configured to handle the message contents.

8. Select Error in the properties dialog navigator and set values for the properties that determine how an error message returned by the Web service is handled.

- If you want the whole Web service error message to be propagated as the output message, leave the *Replace input with Error* check box selected (this is its default setting).

If you want the Web service error message to be included in the output message with part of the input message content, clear this check box and complete the property *Error message location*. If you clear this property, the node copies the input message to the output message and writes the Web service error message over the output message content at the specified location (the input message itself is not modified).

- In *Error message location*, enter the start location within the output message tree at which the parsed elements from the Web service error message bit stream are stored. This property is required only if you have cleared the *Replace input with Error* property.

You can enter any valid ESQL field reference, including expressions within the reference and new field references (to create a new node in the message tree for the response). For example, enter:

```
OutputRoot.XMLNS.ABC.DEF
```

or

```
Environment.WSError
```

If you select the *Replace input with Error* check box, this property is ignored.

9. Select Default in the properties dialog navigator and set values for the properties describing the message domain, message set, message type, and message format that the node uses to determine how to parse the response message returned by the Web service.

If an error message is returned by the Web service, the values of these properties are ignored, and the message is parsed by the BLOB parser.

- In *Message Domain*, select the name of the parser that you are using from the drop-down list. You can choose from the following names:
 - MRM
 - XML
 - XMLNS
 - XMLNSC
 - JMSMap
 - JMSStream
 - MIME
 - BLOB
 - IDOC
- If you are using the MRM or IDOC parser, select the correct message set from the drop-down list in *Message Set*. This list is populated with available message sets when you select MRM or IDOC as the domain.

Leave *Message Set* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
- If you are using the MRM parser, select the correct message from the drop-down list in *Message Type*. This list is populated with messages that are defined in the message set that you have selected.

Leave *Message Type* blank for XML, XMLNS, XMLNSC, JMS, MIME, BLOB, and IDOC parsers.
- If you are using the MRM or IDOC parser, select the format of the message from the drop-down list in *Message Format*. This list includes all the physical formats that you have defined for this message set.

Leave *Message Format* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.

10. Select Validation in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

11. Select General Message Options in the properties dialog navigator. *Parse Timing* is, by default, set to On Demand. This causes validation to be delayed until it is parsed by partial parsing. If you change this to Immediate, partial parsing is overridden and everything in the message is parsed and validated, except those complex types with a Composition of Choice or Message that cannot be resolved at the time. If you change this to Complete, partial parsing is overridden and everything in the message is parsed and validated; complex types with a Composition of Choice or Message that cannot be resolved at the time cause a validation failure.
12. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
13. Click **Apply** to make the changes to the HTTPRequest node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the output terminals to another node:

Connect the out, error, or failure terminal of this node to another node in this message flow to process the message further, process errors, or send the message to an additional destination. If you do not connect the error terminal, the message is discarded. If you do not connect the failure terminal, the broker provides default error processing, described in “Handling errors in message flows” on page 111.

Terminals and properties

The HTTPRequest node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if a failure is detected during processing in the node.
Out	The output terminal to which the message is routed if it represents successful completion of the Web service request, and if further processing is required within this message flow.
Error	The output terminal to which messages that include an HTTP status code that is not in the range 200 through 299, including redirection codes (3xx) if you have not set property <i>Follow HTTP redirection</i> .

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The HTTPRequest node Basic properties are described in the following table.

Property	M	C	Default	Description
Web Service URL	Yes	Yes		The URL for the Web service. You must provide this in the form <code>http://<hostname>[:<port>]/[<path>]</code> where <ul style="list-style-type: none"> <code>http://<hostname></code> must be specified <code><port></code> defaults to 80. If you specify a value, you must include the <code>:</code> before the port number <code><path></code> defaults to <code>/</code>. If you specify a value, you must include the <code>/</code> before the path.
Request Timeout	Yes	No	120	The time in seconds that the node waits for a response from the Web service. The valid range is 1 to $(2^{31})-1$. You cannot enter a value that represents an unlimited wait.
HTTP(S) Proxy Location	No	Yes		The proxy server to which requests are sent. This must be in the form <code>hostname:port</code> .
Follow HTTP(S) redirection.	No	No	Cleared	Whether HTTP redirections are followed. If you select the check box, redirections are followed. If you clear this check box, redirections are not followed.
HTTP Version	No	Yes	1.0	The HTTP version to use for requests. Valid values are 1.0 and 1.1.

Property	M	C	Default	Description
Enable HTTP/1.1 Keep-Alive	No	Yes	Selected (if HTTP Version is 1.1)	Use HTTP/1.1 Keep-Alive

The HTTPRequest node SSL properties are described in the following table.

Property	M	C	Default	Description
Protocol	No	Yes	SSL	The SSL protocol to use when making an HTTPS request.
Allowed SSL Ciphers	No	Yes		A comma-separated list of ciphers to use when making an SSL request. The default value of an empty string means use all available ciphers.

The HTTPRequest node Advanced properties are described in the following table.

Property	M	C	Default	Description
Use whole input message as request	No	No	Selected	Whether the whole input message body is to be passed to the Web service. If you select this check box, this action is performed. If you clear this check box, you must specify <i>Request message location in tree</i> .
Request message location in tree	Yes	No	InputRoot	The start location from which the bit stream is created for sending to the Web service. This property takes the form of an ESQL field reference.
Replace input message with web-service response	No	No	Selected	Whether the Web service response message replaces the copy of the input message as the content of the output message created. If you select this check box, this action is performed. If you clear this check box, you must specify <i>Response message location in tree</i> .
Response message location in tree	Yes	No	OutputRoot	The start location at which the parsed elements from the Web service response bit stream are stored. This property takes the form of an ESQL field reference.
Generate default HTTP headers from input	No	No	Selected	Whether an HTTPRequestHeader is generated. If you select this check box, this action is performed. If you clear this check box, a valid HTTPRequestHeader must exist in the input message.

The HTTPRequest node Error properties are described in the following table.

Property	M	C	Default	Description
Replace Input with Error	No	No	Selected	Whether the input message content is to be replaced by the error message content. If you select this check box, the action is performed. If you clear this check box, you must specify <i>Error message location</i> .
Error message location	Yes	No	OutputRoot	The start location at which the parsed elements from the Web service error bit stream are stored. This property takes the form of an ESQL field reference.

The HTTPRequest node Default properties are described in the following table.

Property	M	C	Default	Description
Message Domain	No	No		The domain that will be used to parse the response message that is received from the Web service.
Message Set	No	No		The name or identifier of the message set in which the response message is defined.
Message Type	No	No		The name of the response message.
Message Format	No	No		The name of the physical format of the response message.

The Validation properties of the HTTPRequest node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	No	Yes	None	Whether validation takes place. Valid values are None, Content and Value, Content, and Inherit.
Failure Action	No	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	No	No	Selected	This property cannot be edited. The default action, which is indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	No	No	None	This property cannot be edited.

The properties of the General Message Options for the HTTPRequest node are described in the following table.

Property	M	C	Default	Description
Parse Timing	No	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to “Parsing on demand” on page 706 for a full description of this property.
Use MQRFH2C Compact Parser for MQRFH2 Domain	No	No	Cleared	This property controls whether the MQRFH2C Compact Parser, instead of the MQRFH2 parser, is used for MQRFH2 headers.

The properties of the XMLNSC Parser Options for the HTTPRequest node are described in the following table.

Property	M	C	Default	Description
Use XMLNSC Compact Parser for XMLNS Domain	No	No	Cleared	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.

Property	M	C	Default	Description
Mixed Content Retain Mode	No	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.
Comments Retain Mode	No	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.
Processing Instructions Retain Mode	No	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The HTTPRequest node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Input node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 541
- “Configuring the Input node” on page 541
- “Terminals and properties” on page 541

Purpose

The Input node provides an in terminal for an embedded message flow (a subflow). You can use a subflow for a common task that can be represented by a sequence of message flow nodes. For example, you can create a subflow to increment or decrement a loop counter, or to provide error processing that is common to a number of message flows.

You must use an Input node to provide the in terminal to a subflow; you cannot use a standard input node (a built-in input node such as MQInput, or a user-defined input node).

When you have started your subflow with an Input node, you can connect it to any in terminal on any message flow node, including an Output node.

You can include one or more Input nodes in a subflow. Each one that you include provides a terminal through which you can introduce messages to the subflow. If you include more than one, the order in which the messages are processed through the subflow cannot be predicted.

The Input node is represented in the workbench by the following icon:



When you select and include a subflow in a message flow, it is represented by the icon:



When you include the subflow in a message flow, this icon exhibits a terminal for each Input node that you include in the subflow, and the name of the terminal (which you can see when you hover over it) matches the name of that instance of the Input node. Give your Input nodes meaningful names that you can easily recognize when you use their corresponding terminal on the subflow node in your message flow.

Using this node in a message flow

Look at the following sample to see how you can use this node:

- Error Handler sample

Configuring the Input node

When you have put an instance of the Input node into a message flow, you can configure it by giving it a name.

Right-click the node in the editor view and select **Properties**. The Description properties of the node are displayed.

Enter a short description, a long description, or both.

Click **Apply** to make the changes to the Input node without closing the properties dialog, or click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Input node terminals are described in the following table.

Terminal	Description
Out	The input terminal that delivers a message to the subflow.

The following table describes the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C

indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Input node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

JavaCompute node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the JavaCompute node” on page 543
- “Terminals and properties” on page 544

Purpose

Use the JavaCompute node to do one of the following tasks:

- Examine an incoming message and, depending on its content, propagate it unchanged to one of the node’s two output terminals; the node behaves in a similar way to a Filter node, but uses Java instead of ESQL to decide which output terminal to use.
- Change part of an incoming message and propagate the changed message to one of the output terminals.
- Create and build a new output message that is totally independent of the input message.

The Java code that is used by the node is stored in an Eclipse Java project.

The JavaCompute node is represented in the workbench by the following icon:



Using this node in a message flow

The JavaCompute node can be used in three ways:

- Use the JavaCompute node to look at a message, and propagate the message to an output terminal based on the message content. The message content is not changed and you have read-only access to the message.
- Use the JavaCompute node to modify a message, and propagate the modified message to an output terminal.
- Use the JavaCompute node to create a new message, and propagate the new message to an output terminal.

Look at the samples in the Samples Gallery in the Message Brokers Toolkit.

Configuring the JavaCompute node

You can configure each instance of the JavaCompute node that occurs in a message flow.

To do this, right-click the node in the editor view. The **Open Java** option is presented. Click this property.

The first time that you do this, a wizard is launched that guides you through the creation of a new Java project and a Java class that contains some skeleton code. This skeleton code is presented in a Java editor.

See “Creating Java code for a JavaCompute node” on page 286 for examples of the skeleton code or template that are provided.

If this is not the first time that this has been done, you are presented with Java code in a Java compute perspective.

To associate an instance of a JavaCompute node with a Java class, right-click the node in the editor view of the message flow and select **Properties** and then **Basic**. The node’s basic properties are displayed.

The JavaCompute node has only one basic property. This is *Java Class*. Enter the name of the Java class that is used in this node. This name must be in the list of JavaCompute node classes that are available in the project references for the message flow project.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

When you have completed your configuration, click **Apply**. This makes the changes to the JavaCompute node without closing the properties dialog.

Alternatively, click **OK**. This makes the changes to the JavaCompute node and closes the properties dialog.

You can click **Cancel** to close the dialog and discard all the changes that you have made to the node.

Specifying Java:

Code Java statements to customize the behavior of the JavaCompute node.

For example, you can customize it to create a new output message or messages, using input message or database content (unchanged or modified), or new data. For example, you might want to modify a value in the input message by adding a value from a database, and store the result in a field in the output message.

Code the Java statements that you want in a Java file that is associated with the JavaCompute node.

If a Java file does not already exist for this node, right-click the JavaCompute node and click **Open Java**. This creates and opens a new Java file in the editor view.

If the file already exists, click the **Browse** button beside the *Java Class* property. This displays the JavaCompute Node Type Selection dialog, which lists the Java

classes that are accessible by this message flow. Select the appropriate Java class and click **OK**. The list of matching types show suitable Java classes when at least one character is entered in the Select field.

Note: All Java classes are shown if you enter `**` in the Select field.

Validating messages:

Set the validation properties to define how the message that is produced by the JavaCompute node is to be validated. Note that these properties do not cause the input message to be validated. It is expected that, if such validation is required, the validation has already been performed by the input node or a preceding Validation node.

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

Terminals and properties

The JavaCompute node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the input message is routed if a failure is detected during the computation. (Even if the Validate property is set, messages propagated to the failure terminal of the node are not validated.)
Out	The output terminal to which the transformed message is routed.
Alternate	An alternative output terminal to which the transformed message can be routed, instead of to the Out terminal.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The JavaCompute node has the Basic property that is described in the following table.

Property	M	C	Default	Description
Java Class	Yes	No	None	The name of the Java class that is used in this node. This name must appear in the list of Java classes that are available to be used.

The Validation properties of the JavaCompute node are described in the following table.

Property	M	C	Default	Description
Validate	Yes	Yes	None	Whether validation takes place, and what part of the message is validated. Valid values are None, Content and Value, Content, and Inherit.

Property	M	C	Default	Description
Failure Action	Yes	No	Exception	What happens if a validation failure occurs. You can set this property only if <i>Validate</i> is set to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that all value constraints are included in the validation.
Fix	Yes	No	None	This property cannot be edited. Minimal fixing is provided. Valid values are None, and Full.

The properties of the General Message Options for the JavaCompute node are described in the following table.

Property	M	C	Default	Description
Parse Timing	Yes	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to “Parsing on demand” on page 706 for a full description of this property.
Use MQRFH2C Compact Parser for MQRFH2 Domain	No	No	False	This property controls whether the MQRFH2C Compact Parser, instead of the MQRFH2 parser, is used for MQRFH2 headers.

The properties of the XMLNSC Parser Options for the JavaCompute node are described in the following table.

Property	M	C	Default	Description
Use XMLNSC Compact Parser for XMLNS Domain	Yes	Cleared	No	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.
Mixed Content Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.
Comments Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.
Processing Instructions Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The Description properties of the JavaCompute node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

JMSInput node

This topic contains the following sections:

- “Purpose”
- “Using the JMSInput node in a message flow”
- “Making the JMS Provider client available to the JMS Nodes” on page 547
- “Configuring the JMSInput node” on page 547
- “Terminals and properties” on page 553

Purpose

Use the JMSInput node to receive messages from JMS destinations. JMS destinations are accessed through a connection to a JMS provider. The JMSInput node acts as a JMS message consumer and can receive all six message types that are defined in the Java Message Service Specification, version 1.1. Messages are received by using method calls, which are described in the JMS specification.

The JMSInput node is represented in the workbench by the following icon:



Using the JMSInput node in a message flow

The JMS Nodes sample contains a message flow in which the JMSInput node is used. Refer to this sample for an example of how to use the JMSInput node.

The JMSInput node receives and propagates messages with a JMS message tree. You can set the properties of the JMSInput node to control the way that the JMS messages are received.

The JMSInput node handles messages in the following message domains:

- BLOB
- XML
- XMLNS
- XMLNSC
- MRM
- JMSMap
- JMSStream
- MIME
- IDOC

Message flows, which handle messages that are received from connections to JMS providers, must always start with a JMSInput node. If you include an output node in a message flow that starts with an JMSInput node, it can be any of the supported output nodes (including user-defined output nodes); you do not have to

include an JMSOutput node. However, if you do not include a JMSOutput node, you must include the JMSMQTransform node to transform the message to the format that is expected by the output node.

If you are propagating JMS messages and creating a message flow to use as a subflow, you cannot use a standard input node; you must use an instance of the JMSInput node as the first node in order to create an In terminal for the subflow.

Restriction: There is currently a restriction when using the JMSInput node to receive publication topics. The node internally restricts the message flow property *Additional Instances* to zero to prevent the receipt of duplicate publications.

Making the JMS Provider client available to the JMS Nodes

For distributed platforms, copy the java .jar files and any native libraries for the JMS provider client into a the broker shared-classes directory. For example, on Windows C:\Documents and Settings\All Users\Application Data\IBM\MQSI\shared-classes. This ensures that the java class path for the JMS nodes is set correctly.

For z/OS, there is no shared-classes directory. Instead you must specify each JMS provider java .jar file in the class path in the BIPPROF member of the broker's PDS (Partitioned Data Set). Then update the LIBPATH with any native libraries, and submit the BIPGEN JCL job to update the broker ENVFILE.

Configuring the JMSInput node

When you have put an instance of the JMSInput node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The basic properties of the node are displayed in the properties dialog.

All mandatory properties that do not have a default value defined are marked with an asterisk on the properties dialog.

Configure the JMSInput node as follows:

1. Select *Basic* in the properties dialog navigator and complete the following properties:
 - Enter an *Initial Context Factory* value. A JMS application uses the initial context to obtain and look up the JNDI administered objects for the JMS provider. The default value is `com.sun.jndi.fscontext.ReffSContextFactory`, which defines the file-based initial context factory for the WebSphere MQ JMS provider. To identify the name of the Initial Context Factory for the JMS provider, refer to the JMS provider documentation.
 - Enter a value for the *Location JNDI Bindings*. This value specifies either the file system path or the LDAP location for the bindings file. The bindings file contains definitions for the JNDI administered objects that are used by the JMSInput node.

When you enter a value for *Location JNDI Bindings*, ensure that it is compliant with the following instructions:

- Construct the bindings file before you deploy a message flow that contains a JMSInput node.
- Do not include the filename of the bindings file in this field.

- If you have specified an LDAP location that requires authentication, you must configure separately both the LDAP principal (userid) and LDAP credentials (password). These values are configured at broker level. For information on configuring these values, refer to the `mqscreatebroker` and `mqschangebroker` commands.
- The string value should include the leading keyword, which is one of the following: `file:/`, `iiop:/`, or `ldap:/`.

For information about constructing the JNDI administered objects bindings file, refer to the documentation that is supplied with the JMS provider.

- Enter a *Connection Factory Name*. The connection factory name is used by the JMSInput node to create a connection to the JMS provider. This name must already exist in the bindings file.
 - Enter a *Backout Destination* name. Input messages are sent to this destination when errors prevent the message flow from processing the message, and the message must be removed from the input destination. The backout destination name must exist in the bindings file.
 - Enter a value for the *Backout Threshold*. This value determines when an input message is put to the *Backout Destination*. For example, if the value is 3, the JMS provider attempts to deliver the message to the input destination three times. After the third attempted delivery, the message is removed from the input destination and is sent to the backout destination. The default value is 0.
2. Select *Default* in the properties dialog navigator and set values for the properties that describe the message domain, message set, message type, and message format.
- In *Message Domain*, select the name of the parser that you are using from the drop-down list. You can choose from the following names:
 - MRM
 - XML
 - XMLNS
 - XMLNSC
 - JMSMap
 - JMSStream
 - MIME
 - BLOB
 - IDOC

If *Message Domain* is left blank, the JMSInput node determines the domain in one of two ways:

- by checking for the presence of data in the JMSType header value of the JMS input message.
- based upon the Java Class of the JMS message.

For information on the order of precedence for determining the message domain, refer to Order of precedence for deriving the message domain.

- If you are using the MRM or IDOC parser, select the correct message set from the drop-down list in *Message Set*. This list is populated with available message sets when you select MRM or IDOC as the domain. Leave *Message Set* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
- If you are using the MRM parser, select the correct message from the drop-down list in *Message Type*. This list is populated with messages that are defined in the message set that you have selected.

Leave *Message Type* blank for XML, XMLNS, XMLNSC, JMS, MIME, BLOB, and IDOC parsers.

- If you are using the MRM or IDOC parser, select the format of the message from the drop-down list in *Message Format*. This list includes all the physical formats that you have defined for this message set.

Leave *Message Format* blank for XML, XMLNS, XMLNSC, JMSMap, JMSStream, MIME, and BLOB parsers.

3. If the JMSInput node is to be used to subscribe to a topic, select *Pub/Sub* in the properties dialog navigator.

- Enter the name of the *Subscription Topic*.
 - If this property is configured, the node operates in the publish/subscribe message domain only.
 - This property is mutually exclusive with *Source Queue* in the *Point to Point* properties section.
 - The *Subscription Topic* name must conform to the standards of the JMS provider that is being used by the node.
- If the node is to receive publications from a durable subscription topic, enter a *Durable Subscription ID*.
 - Removing a durable subscription is a separate administration task. For information on removing a durable subscription, refer to the JMS provider documentation.
 - This property is valid only when a *Subscription Topic* string has been specified.

4. If the JMSInput node is to be used to receive point to point messages, select *Point to Point* in the properties dialog navigator.

- Enter the *Source Queue* name, where the *Source Queue* is the JMS queue that is listed in the bindings file.

5. If filtering of messages is required, select *Message Selectors* in the properties dialog navigator.

- If the JMS provider is required to filter messages based on message properties that are set by the originating JMS client application, enter a value for *Application Property*, specifying both the property name and the selection conditions; for example, `OrderValue > 200`.

Leave this property blank if you do not want the input node to select based upon application property. Refer to JMS message selectors for a description of how to construct the message selector.

- If the JMS provider is required to filter messages that have been generated at specific times, enter a value for *Timestamp*, where the value is an unqualified Java millisecond time; for example, `105757642321`. Qualify the selector with operators such as BETWEEN or AND.

Leave this property blank if you do not want the input node to select based on JMSTimeStamp.

- If the JMS provider is required to filter messages based on the JMSDeliveryMode header value in the JMS messages, select an option for *Delivery Mode* from the drop-down list. You can choose from:

- *Non Persistent* to receive messages marked as non persistent by the originating JMS client application. This is the default option.
- *Persistent* to receive messages marked as persistent by the originating JMS client application.

- If the JMS provider is required to filter messages based upon the JMSPriority header value in the JMS message, enter a value for *Priority*.

Valid values for message priority are from 0 (lowest) to 9 (highest). For example, you can enter 5 to receive messages of priority 5. You can also qualify the selector; for example > 4 to receive messages with a priority greater than 4, or BETWEEN 4 AND 8 to receive messages with a priority in the range 4 to 8.

Leave this property blank if you do not want the input node to select based on JMSPriority.

- If the JMS provider is required to filter messages based upon the JMSMessageID header, enter a value for *Message ID*.

Enter a specific Message ID, or enter a conditional selector; for example, enter > WMBRK123456 to return messages where the Message ID is greater than the WMBRK123456.

Leave this property blank if you do not want the input node to make a selection based on JMSMessageID.

- If the JMS provider is required to filter messages based upon the JMSRedelivered header, enter a value for *Redelivered*.

Enter FALSE if the input node accepts only messages that have not been redelivered by the JMS Provider.

Enter TRUE if the input node accepts only messages that have been redelivered by the JMS Provider.

Leave this property blank if you do not want the input node to select based on JMSRedelivered.

- If the JMS provider is required to filter messages based upon the JMSCorrelationID header, enter a value for *Correlation ID*.

Enter a specific Correlation ID or enter a conditional string; for example, WMBRKABCDEFGH returns messages whose Correlation ID matches this value.

Leave this property blank if you do not want the input node to select based on JMSCorrelationID.

6. Select *Advanced* in the properties dialog navigator.

- To define the transactional characteristics of how the message is handled, select from the *Transaction Mode* drop-down list. You can choose one of the following options:

- Select *none* if the incoming message is to be treated as non persistent. In this case the message will be received using a non transacted JMS Session that is created using the Session.AUTO_ACKNOWLEDGE flag.

- Select *local* if the JMSInput node should coordinate the commit or roll back of JMS messages received by the node, along with any other resources such as DB2 or WebSphere MQ that perform work within the message flow. In this case the node will use a transacted JMS Session.

- Select *global* if the JMSInput node should participate in a global message flow transaction that will be managed by the broker's external syncpoint coordinator. The syncpoint coordinator is the broker's Queue Manager on distributed platforms and RRS (Resource Recovery Services) on z/OS. In this case, any messages received by the node will be globally coordinated using an XA JMS Session.

7. Select *Validation* in the properties dialog navigator to set the node properties that relate to input message validation.

- Select an option from the *Validate* drop-down list. The options that are available are:

- None

- Content and Value

- Content

If you select Content or Content and Value, select an option from the *Failure Action* drop-down list. The options that are available are:

- User Trace
- Local Error Log
- Exception (default value)
- Exception List

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

8. Select General Message Options in the properties dialog navigator. The default value for *Parse Timing* is set to On Demand. This causes validation to be delayed until the message is parsed by partial parsing.

If you change this value to Immediate, partial parsing is overridden and everything in the message is parsed and validated, except complex types with a Composition of Choice or Message that cannot be resolved at the time.

If you change this to Complete, partial parsing is overridden and everything in the message is parsed and validated; complex types with a Composition of Choice or Message that cannot be resolved at the time cause a validation failure.

9. Select *XMLNSC Parser Options* in the properties dialog navigator.

If you want to *Use XMLNSC Compact Parser for XMLNS Domain* select the check box.

For *Mixed Content Retain Mode*, *Comments Retain Mode*, and *Processing Instructions Retain Mode*, the drop-down boxes offer the following choices:

- None
- All

10. Select *Description* in the properties dialog navigator to enter a short description, a long description, or both.

11. Click **Apply** to make the changes to the JMSInput node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

For each message that is received successfully, the JMSInput node routes the message to the out terminal. If this fails, the message is retried. If the retry threshold is reached, where the threshold is defined by the *BackoutThreshold* attribute of the node, the message is routed to the failure terminal.

You can connect nodes to the failure terminal to handle this condition. If you have not connected nodes to the failure terminal, the message is written to the backout destination. If a backout destination has not been provided, an error message is issued and the node stops processing further input. The error message is bip4669.

If the message is caught by the JMSInput node after an exception has been thrown elsewhere in the message flow, the message is routed to the catch terminal. If you have not connected nodes to the catch terminal, the node will backout message for re-delivery until the problem is resolved or the backout threshold is reached.

You must define a backout destination. If you do not define a backout destination, the node issues a bip4669 error message and stops processing further input.

Configuring for coordinated transactions:

When you include a JMSInput node in a message flow, the value that you set for *Transaction Mode* defines whether messages are received under syncpoint.

- If you set it to *global*, the message is received under external syncpoint coordination, that is, within a WebSphere MQ unit of work. Any messages subsequently sent by an output node in the same instance of the message flow are put under syncpoint, unless the output node has explicitly overridden this.
- If you set it to *local*, the message is received under the local syncpoint control of the JMSInput node. Any messages subsequently sent by an output node in the flow are not put under local syncpoint, unless an individual output node has specified that the message must be put under local syncpoint.
- If you set it to *none*, the message is not received under syncpoint. Any messages subsequently sent by an output node in the flow are not put under syncpoint, unless an individual output node has specified that the message must be put under syncpoint.

The JMS provider can supply additional jar files that are required for transactional support. Refer to the JMS provider documentation. For instance, on Distributed (non z/OS) platforms, the WebSphere MQ JMS provider supplies an extra jar file `com.ibm.mqetclient.jar`. This jar must also be added to the broker `shared_classes` directory. Refer to *Making the JMS Provider client available to the JMS Nodes* in this topic.

When messages are to be received under external syncpoint, additional configuration steps are required. These steps need only be applied the first time that a JMSOutput or JMSInput is deployed to the Broker for a particular JMS provider:

- On distributed platforms, the external syncpoint coordinator for the broker is WebSphere MQ. Before you deploy a message flow where the *Transaction Coordination* is set to *Global*, modify the queue manager `.ini` file to include extra definitions for each JMS provider Resource Manager that participates in globally coordinated transactions.

- On Windows, values are configured under the WebSphere MQ Services Resource properties tab. Refer to the *WebSphere MQ System Administration Guide* for more information.

Set the *SwitchFile* property to the following value:

```
<Broker Installation Path>/bin/ JMSSwitch.dll  
XAOpenString=<Initial Context >,<location JNDI>, <Optional>  
ThreadOfControl=THREAD
```

- On distributed platforms (not Windows) add a Stanza to the queue manager “ini” file for each JMS provider. Refer to the *WebSphere MQ System Administration Guide* for more information.

For example,

```
XAResourceManager:  
Name=<Jms_Provider_Name>  
SwitchFile=/<Broker Installation Path>/bin/ JMSSwitch.so  
XAOpenString=<Initial Context >,<location JNDI>, <Optional>  
ThreadOfControl=THREAD
```

Where

name is an installation defined name that identifies a JMS provider Resource Manager.

SwitchFile is the file system path to the JMSSwitch library that is supplied in the bin directory of the broker.

The values for *XAOpenString* are as follows:

- *Initial Context* is the value that is set in the JMSInput node basic property *Initial Context Factory*.
- *location JNDI* is the value that is set in the JMSInput node basic property *Location of JNDI*. This value should include the leading keyword *file:/*, *iio:/* or *ldap:/*

The following parameters are optional:

- *LDAP Principal* which matches the value that is set for the broker by using the *mqsicreatebroker* or *mqsichangebroker* commands.
- *LDAP Credentials* which matches the value that is set for the broker by using the *mqsicreatebroker* or *mqsichangebroker* commands.
- *Recovery Connection Factory Name* which is the JNDI administered connection factory that is defined in the bindings file. If a value is not specified, a default value for *recoverXAQCF* must be added to the bindings file. In either case, the Recovery Connection Factory should be defined as an XA Queue Connection Factory for the JMS provider that is associated with the Initial Context Factor

The optional parameters are comma separated and are positional. Therefore, any parameters that are missing must be represented by a comma.

1. Update the Java CLASSPATH environment variable for the broker's Queue Manager to include a reference to *xarecovery.jar*. For example,
`<Broker Installation Path>/classes/xarecovery.jar`
2. Update the Java PATH environment variable for the broker's Queue Manager to point to the bin directory, which is where the Switch File is located. For example,
`<Broker Installation Path>/bin`

XA cannot use the same Queue Manager for both the broker and the provider until WebSphere MQ Version 5.3, CSD12 and WebSphere MQ Version 6 FixPack 1.

- On z/OS, the external syncpoint manager is Resource Recovery Services (RRS). The only JMS provider that is supported on z/OS is WebSphere MQ JMS. The only Transport option that is supported for WebSphere MQ JMS on z/OS is the Bind option.

Syncpoint control for the JMS provider is managed with RRS syncpoint coordination of the queue manager of the broker. You do not need to modify the .ini file.

Terminals and properties

The terminals of the JMSInput node are described in the following table.

Terminal	Description
Failure	The output terminal to which the message is routed if an error occurs. Even if the Validation property is set, messages propagated to this terminal are not validated.
Out	The output terminal to which the message is routed if it is successfully retrieved from the WebSphere MQ queue.

Terminal	Description
Catch	The output terminal to which the message is routed if an exception is thrown downstream and caught by this node.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Basic properties of the JMSInput node are described in the following table.

Property	M	C	Default	Description
Initial Context Factory	Yes		com.sun.jndi.fscontext.ReffSContextFactory	This is the starting point for a JNDI name space. A JMS application uses the initial context to obtain and look up the connection factory and queue or topic objects for the JMS provider. The default value is that which is used when WebSphere MQ Java is used as the JMS provider.
Location JNDI Bindings	Yes			The system path or the LDAP location for the bindings file.
Connection Factory Name	Yes			The name of the connection factory that is used by the JMSInput node to create a connection to the JMS provider.
Backout Destination	No			The destination that is used by the JMSInput node when a message cannot be processed by the message flow because of errors in the message.
Backout Threshold	No		0	The value that controls when a re-delivered message is put to the backout destination.

The Default properties of the JMSInput node are described in the following table.

Property	M	C	Default	Description
Message Domain	No	No		The domain that will be used to parse the incoming message.
Message Set	No	No		The name or identifier of the message set in which the incoming message is defined.
Message Type	No	No		The name of the incoming message.
Message Format	No	No		The name of the physical format of the incoming message.

The Pub/Sub properties of the JMSInput node are described in the following table.

Property	M	C	Default	Description
Subscription Topic	No	No		The name of the topic to which the node is subscribed.
Durable Subscription ID	No	No		The identifier for a durable subscription topic.

The Point to Point properties of the JMSInput node are described in the following table.

Property	M	C	Default	Description
Source Queue				The name of the queue from which the node receives incoming messages.

The Message Selectors properties of the JMSInput node are described in the following table.

Property	M	C	Default	Description
Application Property	No			Message selector that will filter messages according to the application property value.
Timestamp	No			Message selector that will filter according to the JMSTimestamp.
Delivery Mode	No			Message selector that will filter messages according to the message delivery mode.
Priority	No			Message selector that will filter messages according to the message priority.
Message ID	No			Message selector that will filter messages according to the message ID.
Correlation ID	No			Message selector that will filter messages according to the correlation ID.

The Advanced properties of the JMSInput node are described in the following table.

Property	M	C	Default	Description
Transaction Mode	Yes	No	none	This property is used to determine whether the incoming message is received under external syncpoint, local syncpoint, or out of syncpoint. Valid values are none, local, and global.

The Validation properties of the JMSInput node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	None	This property determines whether validation takes place. Valid values are None, Content, and Content And Value.
Failure Action	Yes	No	Exception	This property determines what happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.

Property	M	C	Default	Description
Fix	Yes	No	None	This property cannot be edited.

The properties of the General Message Options for the JMSInput node are described in the following table.

Property	M	C	Default	Description
Parse Timing	Yes	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to “Parsing on demand” on page 706 for a full description of this property.

The properties of the XMLNSC Parser Options for the JMSInput node are described in the following table.

Property	M	C	Default	Description
Use XMLNSC Compact Parser for XMLNS Domain	Yes	Cleared	No	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.
Mixed Content Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.
Comments Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.
Processing Instructions Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The Description properties of the JMSInput node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

JMSMQTransform node

This topic contains the following sections:

- “Purpose”
- “Using the JMSMQTransform node in a message flow”
- “Terminals and properties”

Purpose

Use the JMSMQTransform node to transform a message with a JMS message tree into a message that has a message tree structure compatible with the format of messages that are produced by the WebSphere MQ JMS provider.

The JMSMQTransform node can be used to send messages to legacy message flows and to interoperate with WebSphere MQ JMS and WebSphere Message Broker publish subscribe.

The JMSMQTransform node handles messages in the following message domains:

- Automatic
- MRM
- XML
- XMLNS
- XMLNSC
- JMSMap
- JMSStream
- BLOB
- MIME
- IDOC

The JMSMQTransform node is represented in the workbench by the following icon:



Using the JMSMQTransform node in a message flow

The JMS Nodes sample contains a message flow in which the JMSMQTransform node is used. Refer to this sample for an example of how to use the JMSMQTransform node.

Terminals and properties

The terminals of the JMSMQTransform node are described in the following table:

Terminal	Description
Failure	The output terminal to which the message is routed if an error occurs. Even if the Validation property is set, messages propagated to this terminal are not validated.
Out	The output terminal to which the message is routed if it is successfully retrieved from the JMS destination.
In	The input terminal that accepts a message for processing by the node.

There are no configurable attributes for this node.

JMSOutput node

This topic contains the following sections:

- “Purpose”
- “Using the JMSOutput node in a message flow”
- “Making the JMS Provider client available to the JMS Nodes”
- “Configuring the JMSOutput node”
- “Terminals and properties” on page 563

Purpose

Use the JMSOutput node to send messages to JMS destinations. The JMSOutput node acts as a JMS message producer and can publish all six message types that are defined in the Java Message Service Specification, version 1.1. Messages are published by using method calls, which are described in the JMS specification.

The JMSOutput node is represented in the workbench by the following icon:



Using the JMSOutput node in a message flow

The JMS Nodes sample contains a message flow in which the JMSOutput node is used. Refer to this sample for an example of how to use the JMSOutput node.

Message flows, which handle messages that are received from connections to JMS providers, must always start with a JMSInput node. If you include the JMSOutput node in a message flow, you do not have to include an JMSInput node. However, if you do not include a JMSInput node, you must include the MQJMSTransform node to transform the message to the format that is expected by the JMSOutput node.

If you are propagating JMS messages and creating a message flow to use as a subflow, you must use an instance of the JMSOutput node as the last node in order to create an out terminal for the subflow.

Making the JMS Provider client available to the JMS Nodes

For distributed platforms, copy the java .jar files and any native libraries for the JMS provider client into a the broker shared-classes directory. For example, on Windows C:\Documents and Settings\All Users\Application Data\IBM\MQSI\shared-classes. This ensures that the java class path for the JMS nodes is set correctly.

For z/OS, there is no shared-classes directory. Instead you must specify each JMS provider java .jar file in the class path in the BIPPROF member of the broker's PDS (Partitioned Data Set). Then update the LIBPATH with any native libraries, and submit the BIPGEN JCL job to update the broker ENVFILE.

Configuring the JMSOutput node

When you have put an instance of the JMSOutput node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed in the properties dialog.

All mandatory properties that do not have a default value defined are marked with an asterisk on the properties dialog.

Configure the JMSOutput node as follows:

1. Select *Basic* in the properties dialog navigator and complete the following properties:
 - Enter an *Initial Context Factory* value. A JMS application uses the initial context to obtain and look up the JNDI administered objects for the JMS provider. The default value is `com.sun.jndi.fscontext.RefFSContextFactory`, which defines the file-based initial context factory for the WebSphere MQ JMS provider.
To identify the name of the Initial Context Factory for the JMS provider, refer to the JMS provider documentation.
 - Enter a value for the *Location JNDI Bindings*. This value specifies either the file system path or the LDAP location for the bindings file. The bindings file contains definitions for the JNDI administered objects that are used by the JMSInput node.
When you enter a value for *Location JNDI Bindings*, ensure that it is compliant with the following instructions:
 - Construct the bindings file before you deploy a message flow that contains a JMSInput node.
 - Do not include the filename of the bindings file in this field.
 - If you have specified an LDAP location that requires authentication, you must configure separately both the LDAP principal (userid) and LDAP credentials (password). These values are configured at broker level. For information on configuring these values, refer to the `mqsicreatebroker` and `mqsichangebroker` commands.
 - The string value should include the leading keyword, which is one of the following: `file:/`, `iiop:/`, or `ldap:/`.
For information about constructing the JNDI administered objects bindings file, refer to the documentation that is supplied with the JMS provider.
 - Enter a *Connection Factory Name*. The connection factory name is used by the JMSInput node to create a connection to the JMS provider. This name must already exist in the bindings file.
2. If the JMSOutput node is to be used to subscribe to a topic, select *Pub/Sub* in the properties dialog navigator.
 - Enter the name of the *Publisher Topic*.
 - If this property is configured, the node operates only in the publish/subscribe message domain.
 - This property is mutually exclusive with *Destination Queue* in the *Point to Point* properties section.
 - The *Publisher Topic* name must conform to the standards of the JMS provider that is being used by the node.
3. If the JMSOutput node is to be used to receive point to point messages then select *Point to Point* in the properties dialog navigator.
 - Enter the *Destination Queue* name for the JMS queue name that is listed in the bindings file.
4. Select *Request* in the properties dialog navigator and complete the following properties:
 - Select an option from the *Destination Mode* drop-down list.

- The default value is *Destination Name*. If this is selected, the message is treated as a request of a datagram and it targets either the *Publication Topic* or the *Destination Queue*.
 - If the message is to be treated as a reply, select *Reply Destination Name*. The JMS provider is supplied with the *JMSReplyTo* value from the *JMSTransport_Header_values* section of the message tree.
 - Enter a value for *Reply To Destination*. You can enter a JMS destination, which can be either a subscription queue or a destination topic. The *Reply To Destination* is the name of the JMS destination to which the receiving application should send a reply message. For a reply message to be returned to this JMS destination, the JMS destination name must be known to the domain of the JMS provider that is used by the receiving client.
The default value is blank, in which case the JMS output message can be regarded as a datagram. If the field is blank, the *JMSOutput* node does not expect a reply from the receiving JMS client.
5. Select *Advanced* in the properties dialog navigator.
- If a *New Correlation ID* is required, select the check box.
 - To define the transactional characteristics of how the message is handled, select from the *Transaction Mode* drop-down list. You can choose one of the following options:
 - Select *none* if the outgoing message is to be treated as non persistent. In this case the message will be sent using a non transacted JMS Session that is created using the *Session.AUTO_ACKNOWLEDGE* flag.
 - Select *local* if the Input node that received the message should coordinate the commit or roll back of JMS messages that have been sent by the *JMSOutput* node, along with any other resources such as DB2 or WebSphere MQ that perform work within the message flow. In this case the node will use a transacted JMS Session.
 - Select *global* if the *JMSOutput* node should participate in a global message flow transaction that will be managed by the broker's external syncpoint coordinator. The syncpoint coordinator is the broker's Queue Manager on distributed platforms and RRS (Resource Recovery Services) on z/OS. In this case, any messages received by the node will be globally coordinated using an XA JMS Session.
 - You can set the persistence of the outgoing JMS message by using the *Delivery Mode* property. Select an option from the drop-down list. You can choose from:
 - *Non Persistent* to indicate to the JMS provider that the message should be treated as non persistent.
 - *Persistent* to mark messages as persistent to the JMS provider and that they should be preserved until successfully delivered to a receiving JMS client application.
 - Enter a value for *Message Expiration* to request that the JMS provider keeps the output JMS message for a specified time.
Enter a value in milliseconds to specify how long the message will be kept by the JMS provider. The default value 0 is used to indicate that the message should not expire.
 - To assign a relative importance to the message, select an option from the *Message Priority* drop-down list. This value can be used for message selection by a receiving JMS client application or a *JMSOutput* node.
Enter a value, where valid values for message priority are from 0 (lowest) to 9 (highest). The default value is 4, which indicates medium priority. Priorities

in the range 0 to 4 relate to normal delivery. Priorities in the range 5 to 9 relate to graduations of expedited delivery.

6. Select *Validation* in the properties dialog navigator to set the node properties that relate to output message validation.

- Select an option from the *Validate* drop-down list. You can choose from:
 - None
 - Content and Value
 - Content
 - Inherit

If you select Content and Value or Content, select a *Failure Action* from the drop-down list. You can choose from:

- User Trace
- Local Error Log
- Exception (default value)
- Exception List

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

7. Select *Description* in the properties dialog navigator to enter a short description, a long description, or both.
8. Click **Apply** to make the changes to the JMSOutput node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

Connect the in terminal of the JMSOutput node to the node from which outbound messages are routed.

Connect the out terminal of this node to another node in the message flow if you want to process the message further, to process errors, or to send the message to an additional destination.

Configuring for coordinated transactions:

When you include a JMSOutput node in a message flow, the value that you set for *Transaction Mode* defines whether messages are received under syncpoint.

- If you set it to *global*, the message is received under external syncpoint coordination, that is, within a WebSphere MQ unit of work. Any messages subsequently sent by an output node in the same instance of the message flow are put under syncpoint, unless the output node has explicitly overridden this.
- If you set it to *local*, the message is sent under the local syncpoint control of the JMSOutput node. Any messages subsequently sent by an output node in the flow are not put under local syncpoint, unless an individual output node has specified that the message must be put under local syncpoint.
- If you set it to *none*, the message is not sent under syncpoint. Any messages subsequently sent by an output node in the flow are not put under syncpoint, unless an individual output node has specified that the message must be put under syncpoint.

The JMS provider can supply additional jar files that are required for transactional support. Refer to the JMS provider documentation. For instance, on Distributed (non z/OS) platforms, the WebSphere MQ JMS provider supplies an extra jar file `com.ibm.mqetclient.jar`. This jar must also be added to the broker `shared_classes` directory. Refer to Making the JMS Provider client available to the JMS Nodes in this topic.

When messages are to be received under external syncpoint, additional configuration steps are required. These steps need only be applied the first time that a JMSOutput or JMSInput is deployed to the Broker for a particular JMS provider:

- On distributed platforms, the external syncpoint coordinator for the broker is WebSphere MQ. Before you deploy a message flow where the *Transaction Coordination* is set to `Global`, modify the queue manager `.ini` file to include extra definitions for each JMS provider Resource Manager that participates in globally coordinated transactions.

- On Windows, values are configured under the WebSphere MQ Services Resource properties tab. Refer to the *WebSphere MQ System Administration Guide* for more information.

Set the *SwitchFile* property to the following value:

```
<Broker Installation Path>/bin/ JMSSwitch.dll
XAOpenString=<Initial Context >,<location JNDI>, <Optional>
ThreadOfControl=THREAD
```

- On distributed platforms (not Windows) add a Stanza to the queue manager “ini” file for each JMS provider. Refer to the *WebSphere MQ System Administration Guide* for more information.

For example,

```
XAResourceManager:
Name=<Jms_Provider_Name>
SwitchFile=/<Broker Installation Path>/bin/ JMSSwitch.so
XAOpenString=<Initial Context >,<location JNDI>, <Optional>
ThreadOfControl=THREAD
```

Where

name is an installation defined name that identifies a JMS provider Resource Manager.

SwitchFile is the file system path to the JMSSwitch library that is supplied in the `bin` directory of the broker.

The values for *XAOpenString* are as follows:

- *Initial Context* is the value that is set in the JMSInput node basic property *Initial Context Factory*.
- *location JNDI* is the value that is set in the JMSInput node basic property *Location of JNDI*. This value should include the leading keyword `file:/`, `iioport:/` or `ldap:/`

The following parameters are optional:

- *LDAP Principal* which matches the value that is set for the broker by using the `mqsicreatebroker` or `mqsichangebroker` commands.
- *LDAP Credentials* which matches the value that is set for the broker by using the `mqsicreatebroker` or `mqsichangebroker` commands.
- *Recovery Connection Factory Name* which is the JNDI administered connection factory that is defined in the `bindings` file. If a value is not specified, a default value for `recoverXAQCF` must be added to the `bindings`

file. In either case, the Recovery Connection Factory should be defined as an XA Queue Connection Factory for the JMS provider that is associated with the Initial Context Factory.

The optional parameters are comma separated and are positional. Therefore, any parameters that are missing must be represented by a comma.

1. Update the Java CLASSPATH environment variable for the broker's Queue Manager to include a reference to xarecovery.jar. For example,


```
<Broker Installation Path>/classes/xarecovery.jar
```
2. Update the Java PATH environment variable for the broker's Queue Manager to point to the bin directory, which is where the Switch File is located. For example,


```
<Broker Installation Path>/bin
```

XA cannot use the same Queue Manager for both the broker and the provider until WebSphere MQ Version 5.3, CSD12 and WebSphere MQ Version 6 FixPack 1.

- On z/OS, the external syncpoint manager is Resource Recovery Services (RRS). The only JMS provider that is supported on z/OS is WebSphere MQ JMS. The only Transport option that is supported for WebSphere MQ JMS on z/OS is the Bind option.

Syncpoint control for the JMS provider is managed with RRS syncpoint coordination of the queue manager of the broker. You do not need to modify the .ini file.

Terminals and properties

The terminals of the JMSOutput node are described in the following table.

Terminal	Description
Failure	The output terminal to which the message is routed if an error occurs. Even if the Validation property is set, messages propagated to this terminal are not validated.
Out	The output terminal to which the message is routed if it is successfully retrieved from the WebSphere MQ queue.
Catch	The output terminal to which the message is routed if an exception is thrown downstream and caught by this node.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Basic properties of the JMSOutput node are described in the following table.

Property	M	C	Default	Description
Initial Context Factory	Yes		com.sun.jndi	<p>fscontext.ReffSContextFactory</p> <p>This is the starting point for a JNDI name space. A JMS application uses the initial context to obtain and look up the connection factory and queue or topic objects for the JMS provider.</p> <p>The default value is that which is used when WebSphere MQ Java is used as the JMS provider.</p>

Property	M	C	Default	Description
Location JNDI Bindings	No			The system path or the LDAP location for the bindings file.
Connection Factory Name	No			The name of the connection factory that is used by the JMSOutput node to create a connection to the JMS provider.

The Pub/Sub properties of the JMSOutput node are described in the following table.

Property	M	C	Default	Description
Publisher Topic	No			The name of the topic from which the node receives published messages.

The Point to Point properties of the JMSOutput node are described in the following table.

Property	M	C	Default	Description
Destination Queue	No			The name of the queue to which the node publishes outgoing messages.

The Request properties of the JMSOutput node are described in the following table.

Property	M	C	Default	Description
Destination Mode	No		Destination Name	This property is used to determine whether the message is to be treated as a request of a datagram or to be treated as a reply.
Reply To Destination	No			This value is the name of the JMS destination to which the receiving application should send a reply message. For a reply message to be returned to this JMS destination, the JMS destination name must be known to the domain of the JMS provider that is used by the receiving client.

The Advanced properties of the JMSOutput node are described in the following table.

Property	M	C	Default	Description
New Correlation ID	No			This property is selected if a <i>New Correlation ID</i> is required.
Transaction Mode	Yes	No	None	This property is to determine whether the incoming message is received under syncpoint. Valid values are None, local, and global.
Delivery Mode	No		Non Persistent	Message selector that will filter messages according to the message delivery mode.
Message Expiration	No		0	This property value is to request that the JMS provider keeps the output JMS message for a specified time. Values are in milliseconds and the default value 0 is used to indicate that the message should not expire.

Property	M	C	Default	Description
Message Priority	No		4	This property value assigns relative importance to the message. This value can be used for message selection by a receiving JMS client application or a JMSOutput node.

The Validation properties of the JMSOutput node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	None	This property determines whether validation takes place. Valid values are None, Content, and Content And Value.
Failure Action	Yes	No	Exception	This property determines what happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	Yes	No	None	This property cannot be edited.

The Description properties of the JMSOutput node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Label node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 566
- “Configuring the Label node” on page 566
- “Terminals and properties” on page 566

Purpose

Use the Label node in combination with a RouteToLabel node to dynamically determine the route that a message takes through the message flow, based on its content. The RouteToLabel node interrogates the LocalEnvironment of the message to determine the identifier of the Label node to which the message must next be routed.

Precede the RouteToLabel node in the message flow with a Compute node that populates the LocalEnvironment of the message with the identifiers of one or more Label nodes that introduce the next sequence of processing for the message.

Design your message flow such that a Label node logically follows a RouteToLabel node within a message flow, but do not physically wire it to the RouteToLabel node. The connection is made by the broker, when required, according to the contents of LocalEnvironment.

The Label node provides a target for a routing decision, and does not process the message it handles in any way. Typically, a Label node connects to a subflow that processes each message in a specific way, and either ends in an output node or in another RouteToLabel node.

The Label node can also be used as the target of a PROPAGATE statement, specified in a Compute or Database node.

The Label node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the Airline Reservations sample to see how you can use this node.

Configuring the Label node

When you have put an instance of the Label node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Label node as follows:

1. There is a single mandatory property for the Label node, *Label Name*, which identifies a target for a RouteToLabel node. *Label Name* must not be the same as the name of the instance of the node itself, and it must be unique within the message flow in which it appears. The name of the instance can be modified by the workbench if the subflow of which this Label node is a part is embedded into another message flow.
2. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
3. Click **Apply** to make the changes to the Label node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Label node terminals are described in the following table.

Terminal	Description
Out	The output terminal to which the message is routed.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Label node Basic properties are described in the following table.

Property	M	C	Default	Description
Label Name	Yes	No		An identifier for the node. It is used as a target for a message routed by a RouteToLabel node.

The Label node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Mapping node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 568
- “Configuring the Mapping node” on page 568
- “Terminals and properties” on page 571

Purpose

Use the Mapping node to construct one or more new messages and populate them with new information, with modified information from the input message, or with information taken from a database. You can modify elements of the message body data, its associated environment, and its exception list.

When you first open or create a message map for the node, if you specify the option **This map is called from a message flow node and maps properties and message body**, the headers in the input message are always copied to the output message without modification.

If you want to modify the message headers in a Mapping node, you must select the option **This map is called from a message flow node and maps properties, headers, and message body**. When you do this, the map that is created allows additional elements, including MQ, HTTP, and JMS headers, to be mapped.

These components of the output message can be defined using mappings that are based on elements of both the input message and data from an external database. You create the mappings associated with this node in the mapping file associated with this node by mapping inputs (message or database) to outputs. You can optionally modify the assignments made by these mappings using supplied or

user-defined functions and procedures: for example you can convert a string value to uppercase when you assign it to the message output field.

Use the Mapping node to:

- Build a new message
- Copy messages between parsers
- Transform a message from one format to another

The Mapping node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following sample to see how you can use this node:

- Pager samples

Configuring the Mapping node

When you have put an instance of the Mapping node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Mapping node as follows:

1. Specify in *Data Source* the name by which the appropriate database is known on the system on which this message flow is to execute. The broker connects to this database with user ID and password information that you have specified on the `mqsicreatebroker`, `mqsichangebroker`, or `mqsisetdbparms` command.
On z/OS systems, the broker uses the broker started task ID, or the user ID and password that were specified on the `mqsisetdbparms` command JCL, `BIPSDBP` in the customization data set `<hlq>.SBIPPROC`.
2. Select the *Transaction* setting from the drop-down menu. The values are:
 - **Automatic** (the default). The message flow, of which the Mapping node is a part, is committed if it is successful. That is, the actions that you define in the mappings are performed and the message continues through the message flow. If the message flow fails, it is rolled back. If you choose **Automatic**, the ability to commit or rollback the action of the Mapping node on the database depends on the success or failure of the entire message flow.
 - **Commit**. If you want to commit any uncommitted actions performed in this message flow on the database connected to this node, irrespective of the success or failure of the message flow as a whole, select **Commit**. The changes to the database are committed even if the message flow fails.
3. In *Mapping Routine*, identify the mapping routine that is to be executed in this node. By default, the name assigned to the mapping routine is identical to the name of the mappings file in which the routine is defined. The default name for the file is the name of the message flow concatenated with the name of the node when you include it in the message flow (for example,

MFlow1_Mapping.mfmap for the first Mapping node in message flow MFlow1). You cannot specify a value that includes spaces.

If you click **Browse** next to this entry field, a dialog is displayed that lists all available mapping routines accessible by this node. Select the routine that you want and click **OK**. The routine name is set in *Mapping Module*.

To work with the mapping routine associated with this node, right-click the node and select **Open Mappings**. If the mapping routine does not exist, it is created for you with the default name in the default file. If the file already exists, you can also open file <flow_name>_<node_name>.mfmap in the Navigator view.

A mapping routine is specific to the type of node with which it is associated; you cannot use a mapping routine that you have developed for a Mapping node with any other node that uses mappings (for example, a DataInsert node). If you create a mapping routine, you cannot call it from any other mapping routine, although you can call it from an ESQL routine.

For more information about working with mapping files, and defining their content, see “Developing message mappings” on page 302.

4. In *Mapping Mode*, specify the mode that you want to use to process information being passed through the Mapping node. You can choose any combination of Message, LocalEnvironment, and Exception components to be generated and modified by the Mapping node.

You must set this property to correctly reflect the output message format that you require. If you select an option (or accept the default value) that does not include a particular component of the message, that component is not included in any output message that is constructed.

(In releases prior to Version 2.1, the associated environment (LocalEnvironment) was known as DestinationList. DestinationList is valid and can be used for compatibility.)

(The Environment component of the message tree is not affected by the mode setting. Its contents, if any, are passed on from this node.)

The options are explained in the table below.

Mode	Description
Message (the default)	The message is generated or passed through by the Mapping node as modified within the node.
LocalEnvironment	The LocalEnvironment tree structure is generated or passed through by the Mapping node as modified within the node.
LocalEnvironment And Message	The LocalEnvironment tree structure and message are generated or passed through by the Mapping node as modified by the node.
Exception	The Exception List is generated or passed through by the Mapping node as modified by the node.
Exception And Message	The Exception List and message are generated or passed through by the Mapping node as modified by the node.
Exception and LocalEnvironment	The Exception List and LocalEnvironment tree structure are generated or passed through by the Mapping node as modified by the node.
All	The message, Exception List, and LocalEnvironment are generated or passed through by the Mapping node as modified by the node.

5. Select Basic in the properties dialog navigator and set or clear the two check boxes:
 - If you want database warning messages to be treated as errors and want the node to propagate the output message to the failure terminal, select the *Treat Warnings as Errors* check box. The box is initially cleared.
When you select the box, the node handles all positive return codes from the database as errors and generates exceptions in the same way as it does for the negative, or more serious, errors.
If you do not select the box, the node treats warnings as normal return codes, and does not raise any exceptions. The most significant warning raised is not found, which can be handled as a normal return code safely in most circumstances.
 - If you want the broker to generate an exception when a database error is detected, select the *Throw Exception on Database Error* check box. The box is initially selected.
If you clear the box, you must handle the error in the message flow to ensure the integrity of the broker and the database: the error is ignored if you do not handle it through your own processing, because you have chosen not to invoke the default error handling by the broker. For example, you could connect the failure terminal to an error processing subroutine.
6. Select Validation in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)
For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.
7. Select General Message Options in the properties dialog navigator. *Parse Timing* is, by default, set to *On Demand*. This causes validation to be delayed until it is parsed by partial parsing. If you change this to *Immediate*, partial parsing is overridden and everything in the message is parsed and validated, except those complex types with a Composition of Choice or Message that cannot be resolved at the time. If you change this to *Complete*, partial parsing is overridden and everything in the message is parsed and validated; complex types with a Composition of Choice or Message that cannot be resolved at the time cause a validation failure.
Select the *Use MQRFH2C Compact Parser for MQRFH2 Domain* check box if you want the MQRFH2C Compact Parser to be used instead of the MQRFH2 parser for MQRFH2 headers.
8. Select the XMLNSCparser options in the properties dialog navigator and select the *Use XMLNSC Compact Parser for XMLNS Domain* check box if you want to use the XMLNSC parser for messages in the XMLNS Domain.
Other properties control whether the XMLNSC parser is used for mixed text, comments and processing instructions in the input message.
9. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
10. Click **Apply** to make the changes to the Mapping node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Mapping node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the input message is propagated if a failure is detected during the computation. If you have selected <i>Treat Warnings as Errors</i> , the node propagates the message to this terminal if database warning messages are returned, even though the processing might have completed successfully.
Out	The output terminal that outputs the message following the execution of the mappings.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Mapping node Basic properties are described in the following table.

Property	M	C	Default	Description
Data Source	No	Yes		The ODBC data source name of the database in which reside the tables to which you refer in the mappings associated with this node (identified by the <i>Mapping Module</i> property).
Transaction	Yes	No	Automatic	The transaction mode for the node. Valid values are Automatic or Commit.
Mapping Routine	Yes	No	Mapping	The name of the mapping routine that contains the statements to execute against the database or the message tree. The routine is unique to this type of node.
Mapping Mode	Yes	No	Message	Select one of the following: <ul style="list-style-type: none"> • Message • LocalEnvironment • LocalEnvironment And Message • Exception • Exception And Message • Exception And LocalEnvironment • All If you want to construct a map that propagates multiple target messages, set this property to Local Environment and Message to ensure the node executes correctly.
Treat Warnings as Errors	Yes	No	Cleared	Treat database SQL warnings as errors. If you select the check box, this action is performed.
Throw Exception on Database Error	Yes	No	Selected	Database errors cause the broker to throw an exception. If you select the check box, this action is performed.

The Validation properties of the Mapping node are described in the following table.

Property	M	C	Default	Description
Validate	Yes	Yes	None	Whether validation takes place. Valid values are None, Content and Value, Content, and Inherit.
Failure Action	Yes	No	Exception	What happens if a validation failure occurs. You can set this property only if <i>Validate</i> is set to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that all value constraints are included in the validation.
Fix	Yes	No	None	This property cannot be edited. Minimal fixing is provided. Valid values are None, and Full.

The properties of the General Message Options for the Mapping node are described in the following table:

Property	M	C	Default	Description
Parse Timing	Yes	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to “Parsing on demand” on page 706 for a full description of this property.
Use MQRFH2C Compact Parser for MQRFH2 Domain	No	No	False	This property controls whether the MQRFH2C Compact Parser, instead of the MQRFH2 parser, is used for MQRFH2 headers.

The XMLNSC parser options for the Mapping node are described in the following table.

Property	M	C	Default	Description
Use XMLNSC Compact Parser for XMLNS Domain	Yes	Cleared	No	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.
Mixed Content Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.
Comments Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.
Processing Instructions Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The Mapping node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

MQeInput node

Attention: The use of message flows that contain MQeInput and MQeOutput nodes in WebSphere Message Broker Version 6.0 is deprecated. The behavior that is described here is intended only for when you are deploying from Version 6.0 to a previous version, and to provide a route for migration. Redesign your flows to remove the MQe nodes and replace them with MQ nodes that are configured to your own specifications and coordinated with your MQe Gateway configuration. For more details see Migrating a message flow that contains WebSphere MQ Everyplace nodes.

This topic contains the following sections:

- “Purpose”
- “Using the MQeInput node in a message flow” on page 574
- “WebSphere MQ Everyplace documentation” on page 574
- “Configuring the MQeInput node” on page 574
- “Terminals and properties” on page 578

Purpose

Use the MQeInput node to receive messages from clients that connect to the broker using the WebSphere MQ Mobile Transport protocol.

The MQeInput node receives messages put to a message flow from a specified bridge queue on the broker’s WebSphere MQ Everyplace queue manager. The node also establishes the processing environment for the messages. You must create and configure the WebSphere MQ Everyplace queue manager before you deploy a message flow containing this node.

Message flows that handle messages received across WebSphere MQ Everyplace connections must always start with an MQeInput node. You can set the MQeInput node’s properties to control the way that messages are received. For example, you can indicate that a message is to be processed under transaction control.

When you deploy message flows containing WebSphere MQ Everyplace nodes to a broker, you must deploy them to a single execution group, regardless of the number of message flows. The WebSphere MQ Everyplace nodes in the flows must all specify the same WebSphere MQ Everyplace queue manager name. You get an error on deploy if you do not meet this restriction.

The MQeInput node handles messages in the following message domains:

- MRM
- XML
- XMLNS
- JMSMap
- JMSStream
- MIME
- BLOB

If you include an output node in a message flow that starts with an MQeInput node, it can be any of the supported output nodes, including user-defined output nodes; you do not have to include an MQeOutput node. You can create a message flow that receives messages from WebSphere MQ Everyplace clients and generates messages for clients that use any of the supported transports to connect to the broker, because you can configure the message flow to request the broker to provide any conversion that is necessary.

WebSphere MQ Everyplace Version 1.2.6 is used by WebSphere Message Broker. This is compatible with later versions of WebSphere MQ Everyplace. Clients using later versions of WebSphere MQ Everyplace, for example Version 2.0, work correctly when connected to this node, although additional functionality not supported in Version 1.2.6 (for example JMS support) does not work.

Queue managers are not interchangeable between different versions of WebSphere MQ Everyplace. Nodes must use a queue manager created using Version 1.2.6. Similarly, the client must use its level of the code when creating a queue manager.

You cannot use MQeInput nodes in message flows that you deploy to z/OS systems.

If you create a message flow to use as a subflow, you cannot use a standard input node: you must use an instance of the Input node as the first node to create an in terminal for the subflow.

If your message flow does not receive messages across WebSphere MQ connections, you can choose one of these other input nodes.

The MQeInput node is represented in the workbench by the following icon:



Using the MQeInput node in a message flow

For an example of how this node can be used, consider a farmer who checks his fields to see how well they are irrigated. He is carrying a PDA device with WebSphere MQ Everyplace installed. He sees an area of field requiring water, so, using his PDA and a Global Satellite Navigation link, he sends a message to an MQeInput node. The data is manipulated using a Compute node, and a message is published by a Publication node so that a remote SCADA device can pick up the message and trigger the irrigation sprinklers. The farmer can see the water delivered to the field minutes after sending his message.

WebSphere MQ Everyplace documentation

You can find further information about WebSphere MQ Everyplace, and the properties of the node, in the WebSphere MQ Everyplace documentation on the WebSphere MQ Web page.

Configuring the MQeInput node

When you have put an instance of the MQeInput node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's default properties are displayed in the properties dialog.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the MQeInput node as follows:

1. Select Default in the properties dialog navigator and set values for the properties that describe the message domain, message set, message type, and message format that the node uses to determine how to parse the incoming message, and the default topic associated with the message.
 - If the incoming message has an MQRFH2 header, you do not have to set values for the Default properties because the values can be derived from the <mcd> folder in the MQRFH2 header. For example:

```
<mcd><Msd>MRM</Msd><Set>DHM4U0906S001</Set><Type>receiptmsg1</Type>
<Fmt>XML</Fmt></mcd>
```

If you set values, and those values differ from those in the MQRFH2 header, the MQRFH2 header values take precedence.
 - In *Message Domain*, select the name of the parser that you are using from the drop-down list. You can choose from the following names:
 - MRM
 - XML
 - XMLNS
 - XMLNSC
 - JMSMap
 - JMSStream
 - MIME
 - BLOB
 - IDOC
 - If you are using the MRM or IDOC parser, select the correct message set from the drop-down list in *Message Set*. This list is populated with available message sets when you select MRM or IDOC as the domain.

Leave *Message Set* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
 - If you are using the MRM parser, select the correct message from the drop-down list in *Message Type*. This list is populated with messages that are defined in the message set that you have selected.

Leave *Message Type* blank for XML, XMLNS, XMLNSC, JMS, MIME, BLOB, and IDOC parsers.
 - If you are using the MRM or IDOC parser, select the format of the message from the drop-down list in *Message Format*. This list includes all the physical formats that you have defined for this message set.

Leave *Message Format* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
 - Enter the message topic in *Topic*. You can enter any characters as the topic name. When messages pass through the MQeInput node, they assume whatever topic name you have entered. (If you are using publish/subscribe, you can subscribe to a topic and see any messages that passed through the MQeInput node under that topic name.)
2. Select General in the properties dialog navigator and complete the following properties:
 - a. Enter the *Queue Name* of the WebSphere MQ Everyplace bridge queue from which this input node retrieves messages. If the queue does not exist, it is created for you when the message flow is deployed to the broker.

- b. Set the level of *Trace* that you want for this node. If trace is active, the trace information is recorded into the file identified by *Trace Filename* (described below). Choose one of:
- None. This is the default setting. No trace output is produced, unless a fatal error occurs.
 - Standard. Minimal trace output is generated to reflect the overall operations of the node.
 - Debug. Trace information is recorded at a level that helps you to debug WebSphere MQ Everyplace programs.
 - Full. All available debug information is recorded to provide a full record of the node activities.

If you set the trace level to *Debug* or *Full*, you will impact the performance of WebSphere MQ Everyplace, and significant trace files can be generated. Use these options for short periods only.

- c. In *Trace Filename*, specify the name of the file to which the trace information is to be written. The directory structure in which the file is specified must already exist: it cannot be created during operation.
- d. Select the *Transaction Mode* to define the transactional characteristics of how this message is handled:
- If you select *Automatic*, the incoming message is received under syncpoint if it is marked persistent; otherwise it is not. The transactionality of any derived messages subsequently sent by an output node is determined by the incoming persistence property, unless the output node has explicitly overridden transactionality.
 - If you select *Yes*, the incoming message is received under syncpoint. Any derived messages subsequently sent by an output node in the same instance of the message flow are sent transactionally, unless the output node has explicitly overridden transactionality.
 - If you select *No*, the incoming message is not received under syncpoint. Any derived messages subsequently sent by an output node in the flow are sent non-transactionally, unless the output node has specified that the message must be put under syncpoint.
- e. The *Use Config File* check box is not selected by default: values for all properties for the MQeInput node are taken from the properties dialog. If you select the check box, the definition of all properties is extracted from the file identified by *Config Filename* (described below) with the exception of the following:
- The *Queue Name* and *Config Filename* general properties
 - All default properties

Use a configuration file only to specify additional properties for the node. If the properties on the properties dialog are sufficient for your needs, do not select the *Use Config File* check box.

- f. If you have selected the *Use Config File* check box, enter the full path and name of the configuration file for WebSphere MQ Everyplace in *Config Filename*. This file must be installed on the system that supports every broker to which this message flow is deployed. If the file does not exist, an error is detected when you deploy the message flow. The default file name is MQeConfig.ini.
- g. In *Queue Manager Name*, specify the name of the WebSphere MQ Everyplace queue manager. This is not related to the queue manager of the broker to which you deploy the message flow containing this node.

Only one WebSphere MQ Everyplace queue manager can be supported. Only one execution group can contain MQeInput or MQeOutput nodes. This property must therefore be set to the same value in every MQeInput node included in every message flow that you deploy to the same broker.

3. Select Channel in the properties dialog navigator and set the maximum number of channels supported by WebSphere MQ Everyplace in *Max Channels*. The default is zero, which means that there is no limit.
4. Select Registry in the properties dialog navigator and complete the following properties:
 - a. Select the type of registry from the drop down list in the *Registry Type* property. You can choose one of the following:
 - File Registry. Registry and security information is provided in the *Directory* specified below.
 - Private Registry. You create the queue manager manually within WebSphere MQ Everyplace, specifying the security parameters that you require.
 - b. In *Directory*, specify the directory in which the registry file is located. This is valid only if you have selected a *Registry Type* of File Registry.
 - c. If you have selected a *Registry Type* of Private Registry, complete the following properties:
 - Specify a *PIN* for the associated queue manager. For further details, refer to the WebSphere MQ Everyplace documentation.
 - Specify a *Certificate Request PIN* for authentication requests. For further details, refer to the WebSphere MQ Everyplace documentation.
 - Provide a *Keyring Password* to be used as a seed for the generation of crypto keys. For further details, refer to the WebSphere MQ Everyplace documentation.
 - In *Certificate Host*, specify the name of the certificate server that WebSphere MQ Everyplace uses for authentication. For further details, refer to the WebSphere MQ Everyplace documentation.
 - In *Certificate Port*, specify the number of the port for the certificate server that WebSphere MQ Everyplace uses for authentication. For further details, refer to the WebSphere MQ Everyplace documentation.
5. Select Listener in the properties dialog navigator and complete the following properties that define the connection type for WebSphere MQ Everyplace:
 - a. In *Listener Type*, select the adapter type that you want to use from the drop down list. The default is Http; you can also select Length or History. For further details, refer to the WebSphere MQ Everyplace documentation.
 - b. In *Hostname*, specify the hostname of the server. Set this to the special value localhost or to the TCP/IP address 127.0.0.1 (the default value), both of which resolve correctly to the hostname of the server to which the message flow is deployed. You can also use any valid hostname or TCP/IP address in your network, but you must use a different message flow for each broker to which you deploy it, or configure this property at deploy time.
 - c. In *Port*, specify the port number on which WebSphere MQ Everyplace is listening. If more than one MQeInput node is included in a message flow deployed to a single broker, each MQeInput node must specify a different number for this property. You must also ensure that the number that you specify does not conflict with other listeners on the broker system, for example, with WebSphere MQ. The default value is 8081.
 - d. In *Time Interval*, specify the timeout value in seconds before idle channels are timed out. The default value is 300 seconds.

Because channels are persistent logical entities that last longer than a single queue manager request, and can survive network breakages, it might be necessary to time out channels that have been inactive for a period of time.

6. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
7. Click **Apply** to make the changes to the MQeInput node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

MQeInput routes each message that it retrieves successfully to the out terminal. If this fails, the message is retried. If the retry timeout expires (as defined by the BackoutThreshold attribute of the input queue), the message is routed to the failure terminal; you can connect nodes to this terminal to handle this condition. If you have not connected the failure terminal, the message is written to the backout queue.

If the message is caught by this node after an exception has been thrown further on in the message flow, the message is routed to the catch terminal. If you have not connected the catch terminal, the message loops continually through the node until the problem is resolved. You must define a backout queue or a dead-letter queue (DLQ) to prevent the message looping continuously through the node.

Configuring for coordinated transactions:

When you include an MQeInput node in a message flow, the value that you set for the *Transaction Mode* property defines whether messages are received under syncpoint:

- If you set it to Yes (the default), the message is received under syncpoint (that is, within a WebSphere MQ unit of work). Any messages subsequently sent by an output node in the same instance of the message flow are put under syncpoint, unless the output node has explicitly overridden this.
- If you set it to Automatic, the message is received under syncpoint if the incoming message is marked persistent. Otherwise, it is not. Any message subsequently sent by an output node is put under syncpoint, as determined by the incoming persistence property, unless the output node has explicitly overridden this.
- If you set it to No, the message is not received under syncpoint. Any messages subsequently sent by an output node in the flow are not put under syncpoint, unless an individual output node has specified that the message should be put under syncpoint.

(The MQOutput node is the only output node that you can configure to override this option.)

Terminals and properties

The MQeInput node terminals are described in the following table.

Terminal	Description
Failure	The output terminal to which the message is routed if an error occurs.

Terminal	Description
Out	The output terminal to which the message is routed if it is successfully retrieved from the WebSphere MQ Everyplace queue.
Catch	The output terminal to which the message is routed if an exception is thrown downstream and caught by this node.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The MQeInput node Default properties are described in the following table.

Property	M	C	Default	Description
Message Domain	No	No		The domain that will be used to parse the incoming message.
Message Set	No	No		The name or identifier of the message set in which the incoming message is defined.
Message Type	No	No		The name of the incoming message.
Message Format	No	No		The name of the physical format of the incoming message.
Topic	No	Yes		The default topic for the input message.

The MQeInput node General properties are described in the following table.

Property	M	C	Default	Description
Queue Name	Yes	Yes		The name of the WebSphere MQ Everyplace bridge queue from which this node retrieves messages for processing by this message flow.
Trace	Yes	No	None	The level of trace required for this node. Valid values are None, Standard, Debug, and Full.
Trace Filename	Yes	Yes	\\MQeTraceFile.trc	The name of the file to which trace records are written.
Transaction Mode	Yes	No	Yes	Whether the incoming message is received under syncpoint. Valid values are Automatic, Yes, and No.
Use Config File	Yes	No	Cleared	Use a configuration file for this node. If you select the check box, this action is performed.
Config Filename	Yes	Yes	\\MQeconfig.ini	The name of the configuration file to be used if the <i>Use Config File</i> check box is selected.
Queue Manager Name	Yes	Yes	ServerQM1	The name of the WebSphere MQ Everyplace queue manager.

The MQeInput node Channel properties are described in the following table.

Property	M	C	Default	Description
Max Channels	Yes	No	0	The maximum number of channels supported by the WebSphere MQ Everyplace queue manager.

The MQeInput node Registry properties are described in the following table.

Property	M	C	Default	Description
Type	Yes	Yes	File Registry	The type of registry information to be used. Valid values are File Registry and Private Registry.
Directory	Yes	Yes	\ServerQM1\registry	The directory in which the registry file exists (valid only if File Registry is selected).
PIN	Yes	Yes		The PIN associated with the WebSphere MQ Everyplace queue manager (valid only if Private Registry is selected).
Certificate Request PIN	Yes	Yes		The PIN used to request authentication (valid only if Private Registry is selected).
Keyring Password	Yes	Yes		The password used to see crypto keys (valid only if Private Registry is selected).
Certificate Host	Yes	Yes		The name of the certificate server (valid only if Private Registry is selected).
Certificate Port	Yes	Yes		The port of the certificate server (valid only if Private Registry is selected).

The MQeInput node Listener properties are described in the following table.

Property	M	C	Default	Description
Listener Type	Yes	Yes	Http	The adapter type for the listener. Valid values are Http, Length, and History.
Hostname	Yes	Yes	127.0.0.1	The hostname of the server.
Port	Yes	Yes	8081	The port on which WebSphere MQ Everyplace listens.
Time Interval	Yes	Yes	300	The WebSphere MQ Everyplace polling interval, specified in seconds.

The MQeInput node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

MQeOutput node

Attention: The use of message flows that contain MQeInput and MQeOutput nodes in WebSphere Message Broker Version 6.0 is deprecated. The behavior that is described here is intended only for when you are deploying from Version 6.0 to a previous version, and to provide a route for migration. Redesign your flows to remove the MQe nodes and replace them with MQ nodes that are configured to your own specifications and coordinated with your MQe Gateway configuration. For more details see Migrating a message flow that contains WebSphere MQ Everyplace nodes.

This topic contains the following sections:

- “Purpose” on page 581

- “Using this node in a message flow”
- “WebSphere MQ Everyplace documentation”
- “Configuring the MQeOutput node”
- “Terminals and properties” on page 583

Purpose

Use the MQeOutput node to send messages to clients that connect to the broker using the WebSphere MQ Mobile Transport protocol.

The MQeOutput node forwards messages to WebSphere MQ Everyplace queue managers. If you specify a non-local destination queue manager, ensure that there is either a route to the queue manager, or store-and-forward queue servicing for the queue manager if it exists.

You cannot use the MQeOutput node to change the transactional characteristics of the message flow. The transactional characteristics set by the message flow’s input node determine the transactional behavior of the flow.

You cannot use MQeOutput nodes in message flows that you deploy to z/OS systems.

If you create a message flow to use as a subflow, you cannot use a standard output node, you must use an instance of the Output node to create an out terminal for the subflow through to propagate which the message.

If you do not want your message flow to send messages to a WebSphere MQ Everyplace queue, you can choose another supported output node.

The MQeOutput node is represented in the workbench by the following icon:



Using this node in a message flow

For an example of how this node can be used, consider a farmer who checks his fields to see how well they are irrigated. He is carrying a PDA device with WebSphere MQ Everyplace installed. He sees that his fields are not being irrigated, so he uses his PDA and a Global Satellite Navigation link to check the water flow valve, and finds that it is faulty. This information is available because the remote SCADA device responsible for controlling the valve has published a diagnostic message, which was retrieved by the broker and forwarded to an MQeOutput node and on to the WebSphere MQ Everyplace client on his PDA.

WebSphere MQ Everyplace documentation

You can find further information about WebSphere MQ Everyplace, and the properties of the node, in the WebSphere MQ Everyplace documentation on the WebSphere MQ Web page.

Configuring the MQeOutput node

When you have put an instance of the MQeOutput node into a message flow you can configure it. Right-click the node in the editor view and click **Properties**. The node’s basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the MQeOutput node as follows:

1. Enter the *Queue Manager Name* and *Queue Name* that specify the destination for the output message if you select *Queue Name* in *Destination Mode* (described below). If you select another option for *Destination Mode*, you do not have to set these properties.
2. Select *Advanced* in the properties dialog navigator and select the *Destination Mode* from the drop-down list. This identifies the queues to which to deliver the output message.
 - *Queue Name*. The message is sent to the queue named in the *Queue Name* property. The properties *Queue Manager Name* and *Queue Name* (on the *Basic* tab) are mandatory if you select this option. This is the default.
 - *Reply To Queue*. The message is sent to the queue named in the *ReplyToQ* field in the MQMD.
 - *Destination List*. The message is sent to the list of queues named in the *LocalEnvironment* (also known as *DestinationList*) associated with the message.
3. Select *Request* in the properties dialog navigator and set properties to define the characteristics of each output message generated.
 - a. Select the *Request* check box to indicate that each output message is marked in the MQMD as a request message (MQMD_REQUEST), and the message identifier field cleared (set to MQMI_NONE) to ensure that WebSphere MQ generates a new identifier. Clear the check box to indicate that each output message is not marked as a request message. You cannot select this check box if you have selected a *Destination Mode* of *Reply To Queue*.
 - b. Enter a WebSphere MQ Everyplace queue manager name in *Reply-to queue manager*. This is inserted into the MQMD of each output message as the reply-to queue manager. This new value overrides the current value in the MQMD.
 - c. Enter a WebSphere MQ Everyplace queue name in *Reply-to queue*. This is inserted into the MQMD of each output message as the reply-to queue. This new value overrides the current value in the MQMD.
4. Select *Description* in the properties dialog navigator to enter a short description, a long description, or both.
5. Click **Apply** to make the changes to the MQeOutput node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

Connect the in terminal to the node from which outbound messages bound are routed.

Connect the out or failure terminal of this node to another node in this message flow if you want to process the message further, process errors, or send the message to an additional destination.

If you do, the LocalEnvironment associated with the message is enhanced with the following information for each destination to which the message has been put by this node:

- Queue name
- Queue manager name
- Message reply identifier (this is set to the same value as message ID)
- Message ID (from the MQMD)
- Correlation ID (from the MQMD)

These values are written in WrittenDestination within the LocalEnvironment tree structure.

If you do not connect either terminal, the LocalEnvironment tree is unchanged.

If you use aggregation in your message flows, you must use these terminals.

Terminals and properties

The MQeOutput node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if a failure is detected when the message is put to the output queue.
Out	The output terminal to which the message is routed if it has been successfully put to the output queue, and if further processing is required within this message flow.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The MQeOutput node Basic properties are described in the following table.

Property	M	C	Default	Description
Queue Manager Name	No	Yes		The name of the WebSphere MQ Everyplace queue manager to which the output queue, specified in <i>Queue Name</i> , is defined.
Queue Name	No	Yes		The name of the WebSphere MQ Everyplace output queue to which this node puts messages.

The MQeOutput node Advanced property is described in the following table.

Property	M	C	Default	Description
Destination Mode	Yes	No	Destination List	The queues to which the output message is sent. Valid values are Queue Name, Reply To Queue, and Destination List.

The MQeOutput node Request properties are described in the following table.

Property	M	C	Default	Description
Request	Yes	No	Cleared	Whether each output message is to be generated as a request message. If you select the check box, the action is performed.
Reply-to queue manager	No	Yes		The name of the queue manager to which the output queue, specified in <i>Reply-to queue</i> , is defined.
Reply-to queue	No	Yes		The name of the reply-to queue to which to put a reply to this request.

The MQeOutput node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

MQGet node

This topic contains the following sections:

- “Purpose”
- “Using the MQGet node in a message flow” on page 585
- “Configuring the MQGet node” on page 585
- “Terminals and properties” on page 590

Purpose

Use the MQGet node to receive messages from clients that connect to the broker using the WebSphere MQ Enterprise Transport, and that use the MQI and AMI application programming interfaces. The MQGet node can also be used to retrieve messages that were previously placed in a WebSphere MQ message queue that is defined to the broker’s queue manager.

The MQGet node reads a message from a specified queue, and establishes the processing environment for the message. If appropriate, you can define the input queue as a WebSphere MQ clustered queue or shared queue.

An MQGet node can be used anywhere within a message flow, unlike an MQInput node which can only be used as the first node in a message flow. The output message tree from an MQGet node is constructed by combining the input tree with the result tree from the MQGET call. You can set the properties of the MQGet node to control the way that messages are received. For example, you can indicate that a message is to be processed under transaction control, or you can request that, when the result tree is being created, data conversion is performed on receipt of every input message.

The MQGet node handles messages in the following message domains:

- MRM
- XML
- XMLNS
- XMLNSC
- JMSMap
- JMSStream
- MIME

- BLOB
- IDOC

The MQGet node is represented in the workbench by the following icon:



Using the MQGet node in a message flow

Look at the following descriptions and samples to see how you can use the MQGet node in a message flow:

- “Using an MQGet node in a request-response flow” on page 407
- SOAP over JMS using MQGet node

Configuring the MQGet node

When you have put an instance of the MQGet node into a message flow, you can configure it.

Right-click the node in the editor view and click **Properties**. The node’s Basic properties are displayed in the properties dialog.

All mandatory properties are marked with an asterisk on the properties dialog.

Configure the MQGet node by doing the following:

1. Enter in *Queue Name* the name of the queue from which the message is to be obtained. You must predefine this WebSphere MQ queue to the queue manager that hosts the broker on which the message flow is deployed. If this queue is not a valid queue, the node generates an exception, and a message is propagated to the failure terminal.
2. Select Default in the properties dialog navigator and set values for the properties that describe the message domain, message set, message type, and message format that the node uses to determine how to parse the incoming message, and the default topic associated with the message.

- If the incoming message has an MQRFH2 header, you do not have to set values for the Default properties because the values can be derived from the <mc> folder in the MQRFH2 header. For example:

```
<mc><Msd>MRM</Msd><Set>DHM4U0906S001</Set><Type>receiptmsg1</Type>
<Fmt>XML</Fmt></mc>
```

If you set values, and those values differ from those in the MQRFH2 header, the values in the MQRFH2 header take precedence.

- In *Message Domain*, select the name of the parser that you are using from the drop-down list. You can choose from the following names:
 - MRM
 - XML
 - XMLNS
 - XMLNSC
 - JMSMap
 - JMSStream
 - MIME
 - BLOB
 - IDOC

- If you are using the MRM or IDOC parser, select the correct message set from the drop-down list in *Message Set*.
Leave *Message Set* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
 - If you are using the MRM parser, select the correct message from the drop-down list in *Message Type*. This list is populated with messages that are defined in the message set that you have selected.
Leave *Message Type* blank for XML, XMLNS, XMLNSC, JMS, IDOC, MIME, and BLOB parsers.
 - If you are using the MRM or IDOC parser, select the format of the message from the drop-down list in *Message Format*. This list includes all the physical formats that you have defined for this message set.
Leave *Message Format* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
3. Select Advanced in the properties dialog navigator and set values for the Advanced properties:
- Select a value for *Transaction Mode* from the drop-down list to define the transactional characteristics of how this message is handled:
 - If you select Automatic, the message is received under syncpoint if it is marked persistent. If the message is not marked as persistent, it is not received under syncpoint. The persistence or non-persistence of the input message determines the transactionality of any derived messages that are subsequently propagated by an output node, unless the output node, or any other subsequent node in the message flow, explicitly overrides the transactionality.
 - If you select Yes, the incoming message is received under syncpoint. Any derived messages that are subsequently propagated by an output node in the same instance of the message flow are sent transactionally, unless the output node, or any other subsequent node in the message flow, has explicitly overridden the transactionality.
 - If you select No, the incoming message is not received under syncpoint. Any derived messages that are subsequently propagated by an output node in the same instance of the message flow are sent non-transactionally, unless the output node, or any other subsequent node in the message flow, has specified that the messages must be put under syncpoint.
 - Select a value for *Generate Mode* from the drop-down list to define which components of the output message are generated within the MQGet node, and which components are taken from the input message.
 - If you select None, all components of the message from the input tree are propagated unchanged.
 - If you select Message, a new Message component is created by the node, but the LocalEnvironment, Environment, and ExceptionList components from the input tree are propagated unchanged. This is the default value for *Generate Mode*.
 - If you select LocalEnvironment, a new LocalEnvironment component is created by the node, but the Message, Environment, and ExceptionList components from the input tree are propagated unchanged.
 - If you select Message and LocalEnvironment, new Message and LocalEnvironment components are created by the node, but the Environment, and ExceptionList components from the input tree are propagated unchanged.

- If you have chosen *Generate Mode* to be either Message or Message and LocalEnvironment, select a value for *Copy Message* from the drop-down list to define which parts of the message are generated within the MQGet node, and which parts are taken from the input message.
 - If you select None, no part of the input message from the input tree is propagated. This is the default value for *Copy Message*.
 - If you select Copy Headers, the headers from the input message in the input tree are copied in.
 - If you select Copy Entire Message, the entire input message from the input tree is copied in.
 - If you have chosen *Generate Mode* to be either LocalEnvironment or Message and LocalEnvironment, select a value for *Copy Local Environment* from the drop-down list to define which parts of the local environment are generated within the MQGet node, and which parts are taken from the input message.
 - If you select None, no part of the local environment is copied in.
 - If you select Copy Entire LocalEnvironment, the entire local environment defined in the input message is copied in. This is the default value for the *Copy Local Environment* property.
 - Provide a value for the *Wait interval* property to specify how many milliseconds to wait for a message to be received from the MQGET call. If you do not provide a value, the default value of 1000 milliseconds is used.
 - Provide a value for the *Minimum message buffer size* property to specify in kilobytes how large the initial buffer for the MQGET call should be. The buffer expands automatically to accept a message of any size, but if it is expected that messages will all be large, by specifying a suitable value you reduce the frequency of the buffer being re-sized. If you do not provide a value, the size of the buffer will initially be 4 kilobytes.
4. Select Validation in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

5. Select Request in the properties dialog navigator and set values for the properties that determine how the request parameters are constructed.
- If the MQMD that is to be used for the MQGET call is not the default location InputRoot.MQMD, specify in *Input MQMD Location* the location of the MQMD.
 - If the location of the parameters for the MQGET call (for example, MQGMO overrides), is not the default location InputLocalEnvironment.MQ.GET, specify the location in *Input MQ Parameters Location*.
 - If you select the *Get by Correlation ID* check box, the CorrelId field of the message to be retrieved must match the CorrelId field in the the *Input MQMD Location*. By default, this check box is cleared.
 - If you select the *Get by Message ID* check box, the MsgId field of the message to be retrieved must match the MsgId field in the the *Input MQMD Location*. By default, this check box is cleared.
 - If you select the *Use complete input MQMD* check box, the entire MQMD of the message to be retrieved must match the value of the *Input MQMD Location* on a bit-for-bit basis. By default, this check box is cleared.
6. Select Result in the properties dialog navigator and set values for the properties that determine how the results of the MQGET call are handled.

- In *Output Data Location*, enter the start location within the output message tree at which the parsed elements from the retrieved message bit string are stored; the default is `OutputRoot`. All elements at this location are deleted, and the default is to replace the input tree message with the retrieved message.

You can enter any valid ESQL field reference - this reference can include expressions - including new field references to create a new node within the message tree for inserting the response into the message that is propagated from the input tree.

For example, `OutputRoot.XMLNS.ABC.DEF` and `Environment.GotReply` are valid field references. See “Using an MQGet node in a request-response flow” on page 407 for more detailed information.

When the retrieved message bit string is parsed to create the contents of the message tree, the message properties that you have specified as the Default properties of the node are used.

- Set a value in *Result Data Location* to control which subtree of the retrieved message is placed in the output message. The default is `ResultRoot` which means that the whole retrieved message is placed in the output message. If, for example, you want only the MQMD from the retrieved message, use `ResultRoot.MQMD`; this subtree is then placed at the location specified by *Output Data Location*.
- Set a value in *Output MQ Parameters Location* to control where the CC (completion code), the RC (return code) and any other MQ parameters (for example the MQMD used by the MQGET call) are placed in the output tree. The default is `OutputLocalEnvironment.MQ.GET`.
- Set a value in *Warning Data Location* to control where the retrieved message is placed when the MQGET call returns a Warning code. The default is `OutputRoot`.

You can enter any valid ESQL field reference (see the description of the *Output Data Location* property). The data that is placed at this location is always the complete result tree, with the body as a BLOB element. *Result Data Location* is not used for warning data.

7. Select General Message Options in the properties dialog navigator. *Parse Timing* is, by default, set to `On Demand`. This causes validation to be delayed until it is parsed by partial parsing.

If you change this to `Immediate`, partial parsing is overridden and everything in the message is parsed and validated, except those complex types with a Composition of Choice or Message that cannot be resolved at the time. If you change this to `Complete`, partial parsing is overridden and everything in the message is parsed and validated; complex types with a Composition of Choice or Message that cannot be resolved at the time cause a validation failure.

For more details refer to “Validation properties for messages in the MRM domain” on page 703.

Select the check box *Use MQRFH2C Compact Parser* if you want the MQRFH2C parser to be used. By default, this check box is cleared which means that the compact parser is not used.

8. Select XMLNSC Parser Options in the properties dialog navigator and set values for the properties that determine how the request parameters are constructed.

For more information, refer to “Manipulating messages using the XMLNSC parser” on page 269.

9. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
10. Click **Apply** to make the changes to the MQGet node without closing the properties dialog, or click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

Connect the Out, Warning, Failure, and No Message output terminals of this node to another node in the message flow if you want to process the message further, process errors, or send the message to an additional destination.

What is propagated to each of the output terminals depends on the condition code (CC) generated by the MQGET call.

If the MQGET call is successful, the MQGet node routes each parsed output message to the Out terminal.

If the MQGET call fails, but with a CC indicating a warning, an unparsed output message is propagated to the Warning terminal.

If the MQGET call fails, with a CC more severe than a warning, the input message is propagated to the Failure terminal.

If the MQGET call fails, with a reason code of `MQRC_NO_MSG_AVAILABLE`, the output message is propagated (without a result body) to the No Message terminal.

If you do not connect the Out, Warning, or No Message terminals to another node in the message flow, anything that is propagated to those terminals is discarded.

If you do not connect the Failure terminal to another node in the message flow, an exception is thrown by the broker when anything is propagated to that terminal.

See “Connecting failure terminals” on page 113 for more information,

Configuring for coordinated transactions:

When you include an MQGet node in a message flow, the value that you set for *Transaction Mode* defines whether messages are received under syncpoint:

- If you set it to Yes (the default), the message is received under syncpoint (that is, within a WebSphere MQ unit of work). Any messages subsequently sent by an output node in the same instance of the message flow are put under syncpoint, unless the output node, or any other subsequent node, has explicitly overridden this.
- If you set it to Automatic, the message is received under syncpoint if the incoming message is marked persistent. Otherwise, it is not. Any message subsequently sent by an output node is put under syncpoint, as determined by the incoming persistence property, unless the output node, or any other subsequent node, has explicitly overridden this.
- If you set it to No, the message is not received under syncpoint. Any messages subsequently sent by an output node in the flow are not put under syncpoint, unless an individual output node, or any other subsequent node, has specified that the message must be put under syncpoint.

Terminals and properties

The terminals of the MQGet node are described in the following table.

Terminal	Description
In	The input terminal that accepts the message that is being processed by the message flow.
Warning	The output terminal to which the output tree is propagated if an error (with a CC that indicates a warning) occurs within the node while trying to get a message from the queue. The MQMD part of the message is parsed, but the rest of the message is an unparsed BLOB element. The warning is discarded if the terminal is not connected, and there is no output propagation from the node at all.
Failure	The output terminal to which the input message is routed if an error (with a CC that indicates an error that is more severe than a warning) occurs within the node while trying to get a message from the queue.
Out	The output terminal to which the message is routed if it is successfully retrieved from the WebSphere MQ queue.
No Message	The output terminal to which the input message is routed if no message was available on the queue. The output message is identical to the input message.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Basic properties of the MQGet node are described in the following table.

Property	M	C	Default	Description
Queue Name	Yes	Yes	None	The name of the WebSphere MQ message queue from which this node retrieves messages.

The Default properties of the MQGet node are described in the following table.

Property	M	C	Default	Description
Message Domain	No	No	None	The domain that will be used to parse the message that is obtained from the message queue.
Message Set	No	No	None	The name or identifier of the message set in which the message that is obtained from the message queue is defined.
Message Type	No	No	None	The name of the message that is obtained from the message queue.
Message Format	No	No	None	The name of the physical format of the message that is obtained from the message queue.

The Advanced properties of the MQGet node are described in the following table.

Property	M	C	Default	Description
Transaction Mode	No	No	Yes	Whether the incoming message is received under syncpoint. Valid values are Automatic, Yes, and No.

Property	M	C	Default	Description
Generate Mode	No	No	Message	Which parts of the message from the input tree are copied. Valid values are Message, LocalEnvironment, Message And LocalEnvironment, and None.
Copy Message	No	No	None	Which parts of the message from the input tree are copied. Valid values are None, Copy Headers, and Copy Entire Message.
Copy Local Environment	No	No		Which parts of the message from the input tree are copied. Valid values are None, and Copy Entire LocalEnvironment. The default value is Copy Entire LocalEnvironment.
Wait interval	Yes	No	1000	The maximum time, in milliseconds, to wait for the message to be obtained from the message queue.
Minimum message buffer size	Yes	No	4	The minimum size, in kilobytes, of the get buffer. The minimum value of this property is 1 kilobyte.

The Validation properties of the MQGet node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	No	Yes	None	Whether validation takes place. Valid values are None, Content, Content and Value, and Inherit.
Failure Action	No	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	No	No	True	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	No	No	None	This property cannot be edited.

The Request properties of the MQGet node are described in the following table.

Property	M	C	Default	Description
Input MQMD Location	No	No		Specifies where in the input message assembly the MQMD that is to be used for the MQGET can be found. The default location is InputRoot.MQMD.
Input MQ Parameters Location	No	No		Specifies where in the input message assembly the MQ parameters (for example, the initial buffer size and the MQGMO overrides) can be found. The default location is InputLocalEnvironment.MQ.GET.
Get by Correlation ID	No	No	False	When selected, this check box causes only messages that have the specified correlation ID to be got.
Get by Message ID	No	No	False	When selected, this check box causes only messages that have the specified message ID to be got.
Use complete input MQMD	No	No	False	When selected, this check box causes the complete MQMD to be used. Otherwise, only the message ID and correlation ID will be used.

The Result properties of the MQGet node are described in the following table.

Property	M	C	Default	Description
Output Data Location	No	No	OutputRoot	Specifies where the output data is placed. If left blank, OutputRoot is used as a default.
Result Data Location	No	No	ResultRoot	Specifies which subtree (of the retrieved message) to use. If left blank, ResultRoot is used as a default, and the whole retrieved message is used. If, for example, ResultRoot.MQMD.ReplyToQ is specified, only that subtree is used.
Output MQ Parameters Location	No	No		Specifies where the output MQ parameters are located. If left blank, OutputLocalEnvironment.MQ.GET is used as a default. Generate Mode should be set to include LocalEnvironment to ensure that the updated values are visible in downstream nodes. The default location is OutputLocalEnvironment.MQ.GET.
Warning Data Location	No	No	OutputRoot	Specifies where the output data is placed if MQGET returns a warning code. If left blank, OutputRoot is used as a default.

The properties of the General Message Options for the MQGet node are described in the following table.

Property	M	C	Default	Description
Parse Timing	No	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to "Parsing on demand" on page 706 for a full description of this property.
Use MQRFH2C Compact Parser for MQRFH2 Domain	No	No	False	This property controls whether the MQRFH2C Compact Parser, instead of the MQRFH2 parser, is used for MQRFH2 headers.

The properties of the XMLNSC Parser Options for the MQGet node are described in the following table.

Property	M	C	Default	Description
Use XMLNSC Compact Parser for XMLNS Domain	No	No	False	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.
Mixed Content Retain Mode	No	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.
Comments Retain Mode	No	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.

Property	M	C	Default	Description
Processing Instructions Retain Mode	No	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The Description properties of the MQGet node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No	Blank	A brief description of the node.
Long Description	No	No	Blank	Text that describes the purpose of the node in the message flow.

MQInput node

This topic contains the following sections:

- “Purpose”
- “Using the MQInput node in a message flow” on page 594
- “Configuring the MQInput node” on page 594
- “Terminals and properties” on page 600

Purpose

Use the MQInput node to receive messages from clients that connect to the broker using the WebSphere MQ Enterprise Transport, and that use the MQI and AMI application programming interfaces.

The MQInput node receives message input to a message flow from a WebSphere MQ message queue defined on the broker’s queue manager. The node uses MQGET to read a message from a specified queue, and establishes the processing environment for the message. If appropriate, you can define the input queue as a WebSphere MQ clustered queue or shared queue.

Message flows that handle messages that are received across WebSphere MQ connections must always start with an MQInput node. You can set the properties of the MQInput node to control the way that messages are received, by causing appropriate MQGET options to be set. For example, you can indicate that a message is to be processed under transaction control. You can also request that data conversion is performed on receipt of every input message.

The MQInput node handles messages in the following message domains:

- MRM
- XML
- XMLNS
- XMLNSC
- JMSMap
- JMSStream
- MIME
- BLOB
- IDOC

If you include an output node in a message flow that starts with an MQInput node, it can be any of the supported output nodes, including user-defined output nodes; you do not have to include an MQOutput node. You can create a message flow that receives messages from WebSphere MQ clients and generates messages for clients that use any of the supported transports to connect to the broker, because you can configure the message flow to request that the broker provides any conversion that is necessary.

If you create a message flow to use as a subflow, you cannot use a standard input node; you must use an instance of the Input node as the first node to create an in terminal for the subflow.

If your message flow does not receive messages across WebSphere MQ connections, you can choose one of the supported input nodes.

The MQInput node is represented in the workbench by the following icon:



Using the MQInput node in a message flow

Look at the following samples to see how you can use the MQInput node:

- Pager samples
- Airline Reservations sample
- Error Handler sample
- Aggregation sample
- JMS Nodes sample
- Large Messaging sample
- Message Routing sample
- Scribble sample
- Soccer Results sample
- Timeout Processing sample
- Video Rental sample
- XMLT sample

Configuring the MQInput node

When you have put an instance of the MQInput node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed in the properties dialog.

All mandatory properties that do not have a default value defined are marked with an asterisk on the properties dialog.

Configure the MQInput node as follows:

1. Enter the name of the queue from which the message flow receives messages. You must predefine this WebSphere MQ queue to the queue manager that hosts the broker to which the message flow is deployed.
2. Select Default in the properties dialog navigator and set values for the properties that describe the message domain, message set, message type, and

message format that the node uses to determine how to parse the incoming message, and the default topic associated with the message.

- If the incoming message has an MQRFH2 header, you do not have to set values for the Default properties because the values can be derived from the <mcd> folder in the MQRFH2 header. For example:

```
<mcd><Msd>MRM</Msd><Set>DHM4U0906S001</Set><Type>receiptmsg1</Type>
<Fmt>XML</Fmt></mcd>
```

If you set values, and those values differ from those in the MQRFH2 header, the values in the MQRFH2 header take precedence.

- In *Message Domain*, select the name of the parser that you are using from the drop-down list. You can choose from the following names:
 - MRM
 - XML
 - XMLNS
 - XMLNSC
 - JMSMap
 - JMSStream
 - MIME
 - BLOB
 - IDOC
 - If you are using the MRM or IDOC parser, select the correct message set from the drop-down list in *Message Set*. This list is populated with available message sets when you select MRM or IDOC as the domain.
Leave *Message Set* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
 - If you are using the MRM parser, select the correct message from the drop-down list in *Message Type*. This list is populated with messages that are defined in the message set that you have selected.
Leave *Message Type* blank for XML, XMLNS, XMLNSC, JMS, MIME, BLOB, and IDOC parsers.
 - If you are using the MRM or IDOC parser, select the format of the message from the drop-down list in *Message Format*. This list includes all the physical formats that you have defined for this message set.
Leave *Message Format* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
 - Enter the message topic in *Topic*. You can enter any characters as the topic name. When messages pass through the MQInput node, they assume whatever topic name you have entered. (If you are using publish/subscribe, you can subscribe to a topic and see any messages that passed through the MQInput node under that topic name.)
3. Select Advanced in the properties dialog navigator to set properties that determine how the message is processed, for example its transactional characteristics. Many of these properties map to options on the MQGET call.
- Select *Transaction Mode* from the drop-down list to define the transactional characteristics of how this message is handled:
 - If you select Automatic, the incoming message is received under syncpoint if it is marked persistent, otherwise it is not. The transactionality of any derived messages subsequently sent by an output node is determined by the incoming persistence property, unless the output node has explicitly overridden transactionality.
 - If you select Yes, the incoming message is received under syncpoint. Any derived messages subsequently sent by an output node in the same

instance of the message flow are sent transactionally, unless the output node has explicitly overridden transactionality.

- If you select *No*, the incoming message is not received under syncpoint. Any derived messages subsequently sent by an output node in the flow are sent non-transactionally, unless the output node has specified that the messages must be put under syncpoint.
- Select *Order Mode* from the drop-down list to determine the order in which messages are retrieved from the input queue. This property has an effect only if the message flow property *Additional Instances* is set to greater than 0, that is, if multiple threads read the input queue. Valid values are:
 - *Default*. Messages are retrieved in the order defined by the queue attributes, but this order is not guaranteed as the messages are processed by the message flow.
 - *By User ID*. Messages that have the same *UserIdentifier* in the MQMD are retrieved and processed in the order defined by the queue attributes; this order is guaranteed to be preserved when the messages are processed. A message associated with a particular *UserIdentifier* that is being processed by one thread is completely processed before the same thread, or another thread, can start to process another message with the same *UserIdentifier*. No other ordering is guaranteed to be preserved.
 - *By Queue Order*. Messages are retrieved and processed by this node in the order defined by the queue attributes; this order is guaranteed to be preserved when the messages are processed. This behavior is identical to the behavior exhibited if the message flow property *Additional Instances* is set to 0.

See “Configuring the node to handle message groups” on page 599 for more details about this option.

- Select the *Logical Order* check box if you want to ensure that messages that are part of a message group are received in the order that has been assigned by the sending application. This option maps to the MQGMO_LOGICAL_ORDER option of the MQGMO of the MQI.

If you clear the check box, messages sent as part of a group are not received in a predetermined order. If a broker expects to receive messages in groups and this check box is not selected, either the order of the input messages is not significant, or the message flow must be designed to process them appropriately.

You must also select the *Commit by Message Group* check box if you want message processing to be committed only after the final message of a group has been received and processed.

More information about the options to which this property maps is available in the *WebSphere MQ Application Programming Reference*.

See “Configuring the node to handle message groups” on page 599 for more details about this option.

- Select the *All Messages Available* check box if you want message retrieval and processing to be done only when all messages in a single group are available. This maps to the MQGMO_ALL_MSGS_AVAILABLE option of the MQGMO of the MQI. Clear this check box if message retrieval does not depend on all messages in a group being available before processing starts.

More information about the options to which this property maps is available in the *WebSphere MQ Application Programming Reference*.

- Enter a message identifier in *Match Message ID* if you want the input node to receive only messages that contain a matching message identifier value in the *MsgId* field of the MQMD.

Enter an even number of hexadecimal digits (characters 0 to 9, A to F, and a to f are valid) up to a maximum of 48 digits. If the ID that you enter is shorter than the size of the *MsgId* field, it is padded on the right with *X'00'* characters. This maps to the *MQMO_MATCH_MSG_ID* option of the MQGMO of the MQI.

Leave this property blank if you do not want the input node to check that the message ID matches.

More information about the options to which this property maps is available in the *WebSphere MQ Application Programming Reference*.

- Enter a message identifier in *Match Correlation ID* if you want the input node to receive only messages that contain a matching correlation identifier value in the *CorrelId* field of the MQMD.

Enter an even number of hexadecimal digits (characters 0 to 9, A to F, and a to f are valid) up to a maximum of 48 digits. If the ID that you enter is shorter than the size of the *CorrelId* field, it is padded on the right with *X'00'* characters. This maps to the *MQMO_MATCH_CORREL_ID* option of the MQGMO of the MQI.

Leave this property blank if you do not want the input node to check that the message ID matches.

More information about the options to which this property maps is available in the *WebSphere MQ Application Programming Reference*.

- Select the *Convert* check box if you want WebSphere MQ to perform data conversion on the message when it is retrieved from the queue.

WebSphere MQ converts the incoming message to the encoding and coded character set specified in the MQMD that the input node supplies on the MQGET call to retrieve the message from the input queue. The message flow generates all its output messages using these values, and puts them to target queues with these *Encoding* and *CodedCharSetID* values set in the MQMD.

This property maps to the *MQGMO_CONVERT* option of the MQGMO of the MQI.

Clear the check box if you do not want WebSphere MQ to convert the message.

If you select this box, you can also specify:

- *Convert Encoding*. Enter the number representing the encoding to which you want to convert numeric data in the message body. Valid values include:
 - 546 for DOS and all Windows systems
 - 273 for all Linux and UNIX systems
 - 785 for z/OS systems

If you do not specify a value, the value in the incoming message MQMD is used.

If you specify an invalid value, no conversion is done.

- *Convert Coded Char Set ID*. Enter the number representing the character set identifier to which you want to convert character data in the message body.

If you do not specify a value, the value in the incoming message MQMD is used.

If you specify an invalid value, no conversion is done.

For more information about WebSphere MQ data conversion, and why you might choose to use this option, see the *WebSphere MQ Application Programming Guide*. For further information about the values that you can specify for *Convert Encoding* and *Convert Coded Char Set ID*, see the *WebSphere MQ Application Programming Reference*.

- Select the *Commit by Message Group* check box if you want message processing to be committed only after the final message of a group has been received and processed. If you leave this check box cleared, a commit is performed after each message has been propagated completely through the message flow.

This property is relevant only if you have selected *Logical Order*.

Set the *Order Mode* property to *By Queue Order* if the messages in a group must be retrieved and processed in the order in which they appear on the queue.

- (z/OS only). Enter a serialization token in *z/OS Serialization Token* if you want to use the serialized access to shared resources that is provided by WebSphere MQ.

The value that you provide for the serialization token must conform to the rules described in the *WebSphere MQ Application Programming Reference*.

For more information about serialization and queue sharing on z/OS, refer to the *WebSphere MQ Concepts and Planning Guide*.

4. Select *Validation* in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

5. Select *General Message Options* in the properties dialog navigator. *Parse Timing* is, by default, set to *On Demand*. This causes validation to be delayed until it is parsed by partial parsing. If you change this to *Immediate*, partial parsing is overridden and everything in the message is parsed and validated, except those complex types with a *Composition of Choice* or *Message* that cannot be resolved at the time. If you change this to *Complete*, partial parsing is overridden and everything in the message is parsed and validated; complex types with a *Composition of Choice* or *Message* that cannot be resolved at the time cause a validation failure.

6. Select *Description* in the properties dialog navigator to enter a short description, a long description, or both.

7. Click **Apply** to make the changes to the MQInput node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

MQInput routes each message that it retrieves successfully to the out terminal. If this fails, the message is retried. If the retry timeout expires (as defined by the *BackoutThreshold* attribute of the input queue), the message is routed to the failure terminal; you can connect nodes to this terminal to handle this condition. If you have not connected the failure terminal, the message is written to the backout queue.

If the message is caught by this node after an exception has been thrown further on in the message flow, the message is routed to the catch terminal. If you have not connected the catch terminal, the message loops continually through the node until the problem is resolved. You must define a backout queue or a dead-letter queue (DLQ) to prevent the message looping continuously through the node.

Configuring for coordinated transactions:

When you include an MQInput node in a message flow, the value that you set for *Transaction Mode* defines whether messages are received under syncpoint:

- If you set it to **Yes** (the default), the message is received under syncpoint (that is, within a WebSphere MQ unit of work). Any messages subsequently sent by an output node in the same instance of the message flow are put under syncpoint, unless the output node has explicitly overridden this.
- If you set it to **Automatic**, the message is received under syncpoint if the incoming message is marked persistent. Otherwise, it is not. Any message subsequently sent by an output node is put under syncpoint, as determined by the incoming persistence property, unless the output node has explicitly overridden this.
- If you set it to **No**, the message is not received under syncpoint. Any messages subsequently sent by an output node in the flow are not put under syncpoint, unless an individual output node has specified that the message must be put under syncpoint.

(The MQOutput node is the only output node that you can configure to override this option.)

Configuring the node to handle message groups:

WebSphere MQ supports message groups; you can specify that a message belongs to a group and that its processing and the processing of all other messages in the group must be handled as one transaction. That is, if the processing on one message in the group fails, all messages in the group are backed out. The message processing is committed when the last message in the group has been processed successfully only if processing of all messages has been successful.

If you include messages in a group, and it is important that all of the messages within the group are read from the queue and processed in the order in which they are defined in the group, you must complete all the actions stated below:

- Select the *Commit by Message Group* check box.
- Select the *Logical Order* check box.
- Set the *Order Mode* to **By Queue Order** or set the message flow property *Additional Instances* to 0. (You can modify message flow properties when you add the message flow to the bar file for deployment.) If you choose either of these options (or both), the message flow processes the messages on a single thread of execution, and a message is processed to completion before the next message is retrieved from the queue. In all other cases, it is possible that multiple threads within a single message flow are processing multiple messages, and there is no guarantee that the final message in a group, which prompts the commit or roll back action, is processed to completion after all other messages in the group.

You must also ensure that you do not have another message flow that is retrieving messages from the same input queue. If you do, there is no guarantee about the order in which the messages within a group are processed.

Terminals and properties

The terminals of the MQInput node are described in the following table.

Terminal	Description
Failure	The output terminal to which the message is routed if an error occurs. Even if the Validation property is set, messages propagated to this terminal are not validated.
Out	The output terminal to which the message is routed if it is successfully retrieved from the WebSphere MQ queue.
Catch	The output terminal to which the message is routed if an exception is thrown downstream and caught by this node.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Basic properties of the MQInput node are described in the following table.

Property	M	C	Default	Description
Queue Name	Yes	Yes		The name of the WebSphere MQ input queue from which this node retrieves messages (using MQGET) for processing by this message flow.

The Default properties of the MQInput node are described in the following table.

Property	M	C	Default	Description
Message Domain	No	No		The domain that will be used to parse the incoming message.
Message Set	No	No		The name or identifier of the message set in which the incoming message is defined.
Message Type	No	No		The name of the incoming message.
Message Format	No	No		The name of the physical format of the incoming message.
Topic	No	Yes		The default topic for the input message.

The Advanced properties of the MQInput node are described in the following table.

Property	M	C	Default	Description
Transaction Mode	Yes	No	Yes	Whether the incoming message is received under syncpoint. Valid values are Automatic, Yes, and No.
Order Mode	Yes	No	Default	The order in which messages are retrieved from the input queue and processed. Valid values are Default, By User ID, and By Queue Order.
Logical Order	Yes	No	Selected	Whether messages are received in logical order, as defined by WebSphere MQ. If you select the check box, this action is performed.

Property	M	C	Default	Description
All Messages Available	Yes	No	Cleared	If you select the check box, all messages in a group must be available before retrieval of a message is possible.
Match Message ID	No	No		A message ID that must match the message ID in the MQMD of the incoming message.
Match Correlation ID	No	No		A correlation ID that must match the correlation ID in the MQMD of the incoming message.
Convert	Yes	No	Cleared	Whether WebSphere MQ converts data in the message to be received, in conformance with the CodedCharSetId and Encoding values set in the MQMD. If you select the check box, this action is performed.
Convert Encoding	No	No		The representation used for numeric values in the message data, expressed as an integer value. This property is valid only if you have selected the <i>Convert</i> check box.
Convert Coded Character Set ID	No	No		The coded character set identifier of character data in the message data, expressed as an integer value. This property is valid only if you have selected the <i>Convert</i> check box.
Commit By Message Group	Yes	No	Cleared	When a transaction is committed when processing messages that are part of a message group. If you select the check box, the transaction is committed when the message group has been processed.
z/OS Serialization Token	No	No		A user-defined token for serialized application support. The value specified must conform to the rules for a valid ConnTag in the WebSphere MQ MQCNO structure. These rules are described in the <i>WebSphere MQ Application Programming Reference</i> .

The Validation properties of the MQInput node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	None	Whether validation takes place. Valid values are None, Content, and Content And Value.
Failure Action	Yes	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	Yes	No	None	This property cannot be edited.

The properties of the General Message Options for the MQInput node are described in the following table.

Property	M	C	Default	Description
Parse Timing	Yes	No	On Demand	This property gives you control over when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to “Parsing on demand” on page 706 for a full description of this property.
Use MQRFH2C Compact Parser for MQRFH2 Domain	Yes	No	No	This property gives you control over whether the MQRFH2C Compact Parser, instead of the MQRFH2 parser, is used for MQRFH2 headers.

The properties of the XMLNSC Parser Options for the MQInput node are described in the following table.

Property	M	C	Default	Description
Use XMLNSC Compact Parser for XMLNS Domain	Yes	Cleared	No	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.
Mixed Content Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.
Comments Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.
Processing Instructions Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The Description properties of the MQInput node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

MQJMSTransform node

This topic contains the following sections:

- “Purpose” on page 603
- “Using the MQJMSTransform node in a message flow” on page 603
- “Terminals and properties” on page 603

Purpose

Use the MQJMSTransform node to receive messages that have a WebSphere MQ JMS provider message tree format, and transform them into a format that is compatible with messages that are to be sent to JMS destinations.

The MQJMSTransform node can be used to send messages to legacy message flows and to interoperate with WebSphere MQ JMS and WebSphere Message Broker publish subscribe.

The JMSMQTransform node handles messages in the following message domains:

- Automatic
- MRM
- XML
- XMLNS
- XMLNSC
- JMSMap
- JMSStream
- BLOB
- MIME
- IDOC

The MQJMSTransform node is represented in the workbench by the following icon:



Using the MQJMSTransform node in a message flow

The JMS Nodes sample contains a message flow in which the MQJMSTransform node is used. Refer to this sample for an example of how to use the MQJMSTransform node.

Terminals and properties

The terminals of the MQJMSTransform node are described in the following table:

Terminal	Description
Failure	The output terminal to which the message is routed if an error occurs. Even if the Validation property is set, messages propagated to this terminal are not validated.
Out	The output terminal to which the message is routed if it is successfully retrieved from the WebSphere MQ queue.
In	The input terminal that accepts a message for processing by the node.

There are no configurable attributes for this node.

MQOptimizedFlow node

The **MQOptimizedFlow** node is a complete message flow that provides a high-performance publish/subscribe message flow. The node supports publishers and subscribers that use Java Message Service (JMS) application programming interfaces and the WebSphere® MQ Enterprise Transport.

To take advantage of any performance gain that this node can provide, you must make sure that you have installed WebSphere MQ Version 5.3 Fix Pack 10 for distributed platforms. Refer to the memo.ptf file for Fix Pack 10 for details of the JMS configuration that is required.

Restriction: The MQOptimizedFlow node cannot be used on z/OS® platforms.

This topic contains the following sections:

- “Purpose”
This is a short introduction to the MQOptimizedFlow node and explains why you might want to use the node.
- “Using this node in a message flow”
This explains how to use the MQOptimizedFlow node.
- “Configuring the MQOptimizedFlow node”
This explains how to configure an MQOptimizedFlow node.
- “Terminals and properties” on page 605
This defines the terminals and the properties that you can configure on the MQOptimizedFlow node.

Purpose

Use the MQOptimizedFlow node to replace a publish/subscribe message flow that consists of an MQInput node connected to a Publication node and that uses the JMS over WebSphere MQ transport.

Use the MQOptimizedFlow node to improve performance, particularly where a single publisher produces a persistent publication for a single subscriber

The MQOptimizedFlow node is represented in the workbench by the following icon:



Using this node in a message flow

Use an MQOptimizedFlow node in a message flow to publish a persistent JMS message to a single subscriber.

Because the MQOptimizedFlow node has no terminals, it cannot be connected to any other message flow node.

Configuring the MQOptimizedFlow node

You must configure each instance of an MQOptimizedFlow node that is present in a message flow.

To do this, right-click the node in the editor view of the message flow and click **Properties**. The Basic properties of the node are displayed.

Specify in the *Queue Name* property the name of the WebSphere MQ input queue from which messages are retrieved.

Select Advanced in the properties dialog navigator, and choose Yes as the value of the *Transaction Mode* property.

Select Description in the properties dialog navigator if you want to give a short description, a long description, or both.

Click Apply to make the changes to the MQOptimizedFlow node without closing the properties dialog.

Click OK to apply the changes and close the properties dialog.

Click Cancel to close the properties dialog and discard all the changes that you have made to the properties.

Terminals and properties

The MQOptimizedFlow node has no terminals. It is a complete message flow and cannot be connected to other message flow nodes to extend the message processing.

The following tables describe the node properties. The column headed M indicates whether the property is mandatory; that is, whether you must enter a value if no default value is defined; an asterisk next to the name of the property in the properties dialog denotes this. The column headed C indicates whether the property is configurable; that is, whether you can change the value in the bar file.

The Basic properties of the MQOptimizedFlow node are described in the following table:

Property	M	C	Default	Description
Queue Name	Yes	Yes	none	The name of the WebSphere MQ input queue from which this node retrieves messages for processing by this message flow.

The Advanced properties of the MQOptimizedFlow node are described in the following table:

Property	M	C	Default	Description
Transaction Mode	Yes	No	Yes	Whether the incoming message is received under syncpoint. Valid values are Automatic, Yes, and No.

The Description properties of the MQOptimizedFlow node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

MQOutput node

This topic contains the following sections:

- “Purpose” on page 606
- “Using this node in a message flow” on page 606
- “Configuring the MQOutput node” on page 607
- “Terminals and properties” on page 610

Purpose

Use the MQOutput node to send messages to clients that connect to the broker using the WebSphere MQ Enterprise Transport and that use the MQI and AMI application programming interfaces.

The MQOutput node delivers an output message from a message flow to a WebSphere MQ queue. The node uses MQPUT to put the message to the destination queue or queues that you specify.

If appropriate, you can define the queue as a WebSphere MQ clustered queue or shared queue. When using a WebSphere MQ clustered queue, leave the queue manager name empty.

You can configure the MQOutput node to put a message to a specific WebSphere MQ queue defined on any queue manager accessible by the broker's queue manager, or to the destinations identified in the LocalEnvironment (also known as the DestinationList) associated with the message.

You can set other properties to control the way in which messages are sent, by causing appropriate MQPUT options to be set. For example, you can request that a message is processed under transaction control. You can also specify that WebSphere MQ can, if appropriate, break the message into segments in the queue manager.

If you create a message flow to use as a subflow, you cannot use a standard output node, you must use an instance of the Output node to create an out terminal for the subflow through which to propagate the message.

If you do not want your message flow to send messages to a WebSphere MQ queue, you can choose another supported output node.

The MQOutput node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following samples to see how you can use this node:

- Pager samples
- Airline Reservations sample
- Error Handler sample
- Aggregation sample
- Large Messaging sample
- Message Routing sample
- Timeout Processing sample
- Video Rental sample
- XMLT sample

For an example of how you can use this node, assume that you have written a publishing application that publishes stock updates on a regular basis. The application sends the messages to the broker on an MQInput node, and the message flow makes the publications available to multiple subscribers through a

Publication node. You configure a Compute node to create a new output message whenever one particular stock is changed, and wire this to an MQOutput node to record each price change for this stock.

Configuring the MQOutput node

When you have put an instance of the MQOutput node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the MQOutput node as follows:

1. If you want to send the output message to a single destination queue that is defined by this node, enter the name of the queue to which the message flow sends messages in *Queue Name*. Enter the name of the queue manager to which this queue is defined in *Queue Manager Name*. You must set these properties if you set the Advanced property *Destination Mode* (described below) to Queue Name. If you set *Destination Mode* to another value, these properties are ignored.
2. Select Advanced in the properties dialog navigator. These properties define the transactional control for the message and the way that the message is put to the queue. Many of these properties map to options on the MQPUT call.
 - Select the *Destination Mode* from the drop-down list. This identifies the queues to which the output message is put.
 - Queue Name. The message is sent to the queue named in the *Queue Name* property. The properties *Queue Manager Name* and *Queue Name* (on the Basic tab) are mandatory if you select this option. This is the default.
 - Reply To Queue. The message is sent to the queue named in the ReplyToQ field in the MQMD.
 - Destination List. The message is sent to the list of queues named in the LocalEnvironment (also known as DestinationList) associated with the message.
 - Select the *Transaction Mode* from the drop-down list to determine how the message is put.
 - If you select Automatic (the default), the message transactionality is derived from the way that it was specified at the input node.
 - If you select Yes, the message is put transactionally.
 - If you select No, the message is put non-transactionally.

See “Configuring for coordinated transactions” on page 609 for more information.

 - Select the *Persistence Mode* from the drop-down list to determine whether the message is put persistently.
 - If you select Automatic (the default), the persistence is as specified in the incoming message.
 - If you select Yes, the message is put persistently.
 - If you select No, the message is put non-persistently.
 - If you select As Defined for Queue, the message persistence is set as defined for the WebSphere MQ queue.
 - Select the *New Message ID* check box to generate a new message ID for this message. This maps to the MQPMO_NEW_MSG_ID option of the MQPMO of the MQI.

Clear the check box if you do not want to generate a new ID. Note that a new message ID is still generated if you select the *Request* check box in the Request panel of the properties dialog.

More information about the options to which this property maps is available in the *WebSphere MQ Application Programming Reference*.

- Select the *New Correlation ID* check box to generate a new correlation ID for this message. This maps to the MQPMO_NEW_CORREL_ID option of the MQPMO of the MQI. Clear the check box if you do not want to generate a new ID.

More information about the options to which this property maps is available in the *WebSphere MQ Application Programming Reference*.

- Select the *Segmentation Allowed* check box if you want WebSphere MQ to segment the message within the queue manager when appropriate. You must also set MQMF_SEGMENTATION_ALLOWED in the MsgFlags field in the MQMD for segmentation to occur. Clear the check box if you do not want segmentation.

More information about the options to which this property maps is available in the *WebSphere MQ Application Programming Reference*.

- Select the *Message Context* to indicate how origin context is to be handled. Choose one of the following options:
 - Pass All (maps to the MQPMO_PASS_ALL_CONTEXT option of the MQPMO of the MQI.)
 - Pass Identity (maps to the MQPMO_PASS_IDENTITY_CONTEXT option of the MQPMO of the MQI.)
 - Set All (maps to the MQPMO_SET_ALL_CONTEXT option of the MQPMO of the MQI.)
 - Set Identity (maps to the MQPMO_SET_IDENTITY_CONTEXT option of the MQPMO of the MQI.)
 - Default (maps to the MQPMO_DEFAULT_CONTEXT option of the MQPMO of the MQI.)
 - None (maps to the MQPMO_NO_CONTEXT option of the MQPMO of the MQI.)

More information about the options to which these properties map is available in the *WebSphere MQ Application Programming Reference*.

- Select the *Alternate User Authority* check box if you want the MQOO_ALTERNATE_USER_AUTHORITY option set in the open options (MQOO) of the MQI. If you select this box, this option is specified when the queue is opened for output. The alternate user information is retrieved from the context information in the message. Clear the check box if you do not want to specify alternate user authority. If you clear the box, the broker service user ID is used when the message is put.

3. Select Request in the properties dialog navigator and set the properties to define the characteristics of each output message generated.

- Select the *Request* check box to mark each output message in the MQMD as a request message (MQMT_REQUEST), and clear the message identifier field (set to MQMI_NONE) to ensure that WebSphere MQ generates a new identifier. Clear the check box to indicate that each output message is not marked as a request message. You cannot select this check box if you have selected a *Destination Mode of Reply To Queue*.

Note that a new message identifier is generated even if the *New Message ID* check box is not selected in the Advanced panel of the properties dialog navigator.

- Enter a queue manager name in *Reply-to Queue Manager*. This is inserted into the MQMD of each output message as the reply-to queue manager.
 - Enter a queue name in *Reply-to Queue*. This is inserted into the MQMD of each output message as the reply-to queue.
4. Select **Validation** in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)
For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.
 5. Select **Description** in the properties dialog navigator to enter a short description, a long description, or both.
 6. Click **Apply** to make the changes to the MQOutput node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

Connect the in terminal to the node from which outbound messages bound are routed.

Connect the out or failure terminal of this node to another node in this message flow if you want to process the message further, process errors, or send the message to an additional destination.

If you connect one of these output terminals to another node in the message flow, the LocalEnvironment associated with the message is enhanced with the following information for each destination to which the message has been put by this node:

- Queue name
- Queue manager name
- Message reply identifier (this is set to the same value as message ID)
- Message ID (from the MQMD)
- Correlation ID (from the MQMD)

These values are written in WrittenDestination within the LocalEnvironment tree structure.

If you do not connect either terminal, the LocalEnvironment tree is unchanged.

If you use aggregation in your message flows, you must use the out terminals.

Configuring for coordinated transactions:

When you define an MQOutput node, the option that you select for the *Transaction Mode* property defines whether the message is written under syncpoint:

- If **Yes**, the message is written under syncpoint (that is, within a WebSphere MQ unit of work).
- If **Automatic** (the default), the message is written under syncpoint if the incoming input message is marked persistent.
- If **No**, the message is not written under syncpoint.

Another property of the MQOutput node, *Persistence Mode*, defines whether the output message is marked as persistent when it is put to the output queue:

- If Yes, the message is marked as persistent.
- If Automatic (the default), the message persistence is determined from the properties of the incoming message, as set in the MQMD (the WebSphere MQ message descriptor).
- If No, the message is not marked as persistent.
- If As Defined for Queue, the message persistence is set as defined in the WebSphere MQ queue by the MQOutput node specifying the MQPER_PERSISTENCE_AS_Q_DEF option in the MQMD.

Terminals and properties

The MQOutput node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if a failure is detected when the message is put to the output queue.
Out	The output terminal to which the message is routed if it has been successfully put to the output queue, and if further processing is required within this message flow.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The MQOutput node Basic properties are described in the following table.

Property	M	C	Default	Description
Queue Manager Name	No	Yes		The name of the WebSphere MQ queue manager to which the output queue, specified in <i>Queue Name</i> , is defined.
Queue Name	No	Yes		The name of the WebSphere MQ output queue to which this node puts messages (using MQPUT).

The MQOutput node Advanced properties are described in the following table.

Property	M	C	Default	Description
Destination Mode	Yes	No	Queue Name	The queues to which the output message is sent. Valid values are Destination List, Reply To Queue, and Queue Name.
Transaction Mode	Yes	No	Automatic	Whether the message is put transactionally. Valid values are Automatic, Yes, and No.
Persistence Mode	Yes	No	Automatic	Whether the message is put persistently. Valid values are Automatic, Yes, No, and As Defined for Queue.
New Message ID	Yes	No	Cleared	Whether WebSphere MQ generates a new message identifier to replace the contents of the MsgId field in the MQMD. If you select the check box, this action is performed.

Property	M	C	Default	Description
New Correlation ID	Yes	No	Cleared	Whether WebSphere MQ generates a new correlation identifier to replace the contents of the CorrelId field in the MQMD. If you select the check box, this action is performed.
Segmentation Allowed	Yes	No	Cleared	If appropriate, WebSphere MQ breaks the message into segments in the queue manager. If you select the check box, this action is performed.
Message Context	Yes	No	Pass All	How to handle origin context. Valid values are Pass All, Pass Identity, Set All, Set Identity, and Default.
Alternate User Authority	Yes	No	Cleared	Whether alternate authority is used when the output message is put. If you select the check box, this action is performed.

The MQOutput node Request properties are described in the following table.

Property	M	C	Default	Description
Request	Yes	No	Cleared	Whether to generate each output message as a request message. If you select the check box, this action is performed.
Reply-to Queue Manager	No	Yes		The name of the WebSphere MQ queue manager to which the output queue, specified in <i>Reply-to Queue</i> , is defined.
Reply-to Queue	No	Yes		The name of the WebSphere MQ queue to which to put a reply to this request.

The Validation properties of the MQOutput node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	Inherit	Whether validation takes place. Valid values are None, Content and Value, Content, and Inherit.
Failure Action	Yes	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	Yes	No	None	This property cannot be edited.

The MQOutput node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.

Property	M	C	Default	Description
Long Description	No	No		Text that describes the purpose of the node in the message flow.

MQReply node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the MQReply node”
- “Terminals and properties” on page 614

Purpose

Use the MQReply node to send a response to the originator of the input message. MQReply is a specialized form of the MQOutput node that puts the output message to the WebSphere MQ queue identified by the ReplyToQ field of the input message header. If appropriate, you can define the queue as a WebSphere MQ clustered queue or shared queue.

The MQReply node honors the options set in the Report field in the MQMD. By default (if no options are set), the MQReply node generates a new MsgID and CorrelID in the reply message. If the receiving application expects other values in these fields you must ensure either that the application that puts the message to the message flow input queue sets the required report options, or that you set the appropriate options within the MQMD during message processing in the message flow. For example, use a Compute node to set the Report options in the message.

You can find more information about the Report field in the *WebSphere MQ Application Programming Reference*.

The MQReply node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following sample to see how you can use this node:

- Airline Reservations sample

You might find it appropriate to use this node when receiving an order from a customer. When the order message is processed, a response is sent to the customer acknowledging receipt of the order and providing a possible date for delivery.

Configuring the MQReply node

When you have put an instance of the MQReply node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node’s basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the MQReply node as follows:

1. Select the *Segmentation Allowed* check box if you want WebSphere MQ to break the message into segments in the queue manager, if appropriate. You must also set `MQMF_SEGMENTATION_ALLOWED` in the `MsgFlags` field in the MQMD for segmentation to occur.

More information about the options to which this property maps is available in the *WebSphere MQ Application Programming Reference*.

2. Select the *Persistence Mode* that you want for the output message.
 - If you select *Automatic* (the default), the persistence is as specified in the incoming message.
 - If you select *Yes*, the message is put persistently.
 - If you select *No*, the message is put non-persistently.
 - If you select *As Defined for Queue*, the message persistence is set as defined in the WebSphere MQ queue.
3. Select the *Transaction Mode* that you want for the output message.
 - If you select *Automatic* (the default), the message transactionality is derived from how it was specified at the MQInput node.
 - If you select *Yes*, the message is put transactionally.
 - If you select *No*, the message is put non-transactionally.
4. Select *Validation* in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

5. Select *Description* in the properties dialog navigator to enter a short description, a long description, or both.
6. Click **Apply** to make the changes to the MQReply node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

The reply message is put (using MQPUT) to the queue named in the input message MQMD as the ReplyTo queue. You cannot change this destination.

Connecting the output terminals to another node:

Connect the out or failure terminal of this node to another node in this message flow if you want to process the message further, process errors, or send the message to an additional destination.

If you connect one of these output terminals to another node in the message flow, the `LocalEnvironment` associated with the message is enhanced with the following information for each destination to which the message has been put:

- Queue name
- Queue manager name
- Message reply identifier (this is set to the same value as message ID)
- Message ID (from the MQMD)
- Correlation ID (from the MQMD)

These values are written in `WrittenDestination` within the `LocalEnvironment` tree structure.

If you do not connect one of these out terminals, the `LocalEnvironment` tree is unchanged.

If you use aggregation in your message flows, you must use these out terminals.

Configuring for coordinated transactions:

When you define an `MQReply` node, the option that you select for the `Transaction Mode` property defines whether the message is written under syncpoint:

- If `Yes`, the message is written under syncpoint (that is, within a WebSphere MQ unit of work).
- If `Automatic` (the default), the message is written under syncpoint if the incoming input message is marked persistent.
- If `No`, the message is not written under syncpoint.

Another property of the `MQReply` node, `Persistence Mode`, defines whether the output message is marked as persistent when it is put to the output queue:

- If `Yes`, the message is marked as persistent.
- If `Automatic` (the default), the message persistence is determined by the properties of the incoming message, as set in the MQMD (the WebSphere MQ message descriptor).
- If `No`, the message is not marked as persistent.
- If `As Defined for Queue`, the message persistence is set as defined in the WebSphere MQ queue by the `MQReply` node specifying the `MQPER_PERSISTENCE_AS_Q_DEF` option in the MQMD.

Terminals and properties

The `MQReply` node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if a failure is detected when the message is put to the output queue.
Out	The output terminal to which the message is routed if it has been successfully put to the output queue, and if further processing is required within this message flow.

The following tables describe the node properties; the column headed `M` indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed `C` indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The `MQReply` node Advanced properties are described in the following table.

Property	M	C	Default	Description
Segmentation Allowed	Yes	No	Cleared	If appropriate, WebSphere MQ breaks the message into segments in the queue manager. If you select the check box, this action is performed.

Property	M	C	Default	Description
Persistence Mode	Yes	No	Automatic	Whether the message is put persistently. Valid values are Automatic, Yes, No, and As Defined for Queue.
Transaction Mode	Yes	No	Automatic	Whether the message is put transactionally. Valid values are Automatic, Yes, and No.

The Validation properties of the MQReply node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	Inherit	Whether validation takes place. Valid values are None, Content and Value, Content, and Inherit.
Failure Action	Yes	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	Yes	No	None	This property cannot be edited.

The MQReply node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

The MQReply node also has the following properties that you cannot access or modify through the workbench interface. However, these values are used by the broker when the message is processed in the message flow.

Property	Description
Queue Manager Name	The name of the WebSphere MQ queue manager to which the output queue, identified in <i>Queue Name</i> , is defined. This name is retrieved from the ReplyTo field of the MQMD of the input message.
Queue Name	The name of the WebSphere MQ queue to which the output message is put. This name is retrieved from the ReplyTo field of the MQMD of the input message.
Destination	This property always has the value reply.

Output node

This topic contains the following sections:

- “Purpose” on page 616
- “Using this node in a message flow” on page 616
- “Configuring the Output node” on page 616

- “Terminals and properties” on page 617

Purpose

The Output node provides an out terminal for an embedded message flow (a subflow). You can use a subflow for a common task that can be represented by a sequence of message flow nodes. For example, you can create a subflow to increment or decrement a loop counter, or to provide error processing that is common to a number of message flows.

You must use an Output node to provide the out terminal to a subflow; you cannot use a standard output node (a built-in output node such as MQOutput, or a user-defined output node).

You can include one or more Output nodes in a subflow. Each one that you include provides a terminal through which you can propagate messages to subsequent nodes in the message flow in which you include the subflow.

The Output node is represented in the workbench by the following icon:



When you select and include a subflow in a message flow, it is represented by the icon:



When you include the subflow in a message flow, this icon exhibits a terminal for each Output node that you included in the subflow, and the name of the terminal (which you can see when you hover over it) matches the name of that instance of the Output node. Give your Output nodes meaningful names, you can easily recognize them when you use their corresponding terminal on the subflow node in your message flow.

Using this node in a message flow

Look at the following sample to see how you can use this node:

- Error Handler sample

Configuring the Output node

When you have put an instance of the Output node into a message flow, you can configure it by giving it a name.

Right-click the node in the editor view and select **Properties**. The Description properties of the node are displayed.

Enter a short description, a long description, or both.

Click **Apply** to make the changes to the Input node without closing the properties dialog, or click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Output node terminals are described in the following table.

Terminal	Description
In	The output terminal that defines an out terminal for the subflow.

The following table describes the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Output node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Passthrough node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the Passthrough node” on page 618
- “Terminals and properties” on page 618

Purpose

Use the Passthrough node in a subflow as the first node that follows the Input node to identify the subflow in which it is included. You can specify an identifier (Label) in whatever way meets your requirements, for example to identify the level or version of the flow in which it is configured.

The Passthrough node does not process the message in any way. The message that it propagates on its out terminal is the same message that it received on its in terminal.

The Passthrough node is represented in the workbench by the following icon:



Using this node in a message flow

Use this node to identify a subflow. For example, if you develop an error processing subflow to include in several message flows, you might want to modify that subflow. However, you might want to introduce the modified version initially

to just a subset of the message flows in which it is included. Set a value for the instance of the Passthrough node that identifies which version of the subflow you have included.

Configuring the Passthrough node

When you have put an instance of the Passthrough node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Passthrough node as follows:

1. Specify in *Label* the identifier for this node. Enter a value that defines a unique characteristic, for example the version of the subflow in which the node is included.
2. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
3. Click **Apply** to make the changes to the Passthrough node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Passthrough node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Out	The input terminal that delivers a message to the subflow.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Passthrough node Basic properties are described in the following table.

Property	M	C	Default	Description
Label	No	No		The label (identifier) of the node.

The Passthrough node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Publication node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the Publication node” on page 620
- “Terminals and properties” on page 620

Purpose

Use the Publication node to filter output messages from a message flow and transmit them to subscribers who have registered an interest in a particular set of topics. The Publication node must always be an output node of a message flow and has no output terminals of its own.

Use the Publication node (or a user-defined node that provides a similar service) if your message flow supports publish/subscribe applications. Applications expecting to receive publications must register a subscription with a broker, and can optionally qualify the publications that they get by providing restrictive criteria (such as a specific publication topic).

If your subscriber applications use the WebSphere MQ Enterprise Transport to connect to the broker, you can define the queues to which messages are published as WebSphere MQ clustered queues or shared queues.

Publications can also be sent to subscribers within a WebSphere MQ cluster if a cluster queue is nominated as the subscriber queue. In this case, the subscriber should use the name of an “imaginary” queue manager that is associated with the cluster, and should ensure that a corresponding blank queue manager alias definition for this queue manager is made on the broker that satisfies the subscription.

The Publication node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following samples to see how you can use this node:

- Soccer Results sample
- Scribble sample
- JMS Nodes sample
- Pager samples

For an example of how you can use this node, assume that you have written a publishing application that publishes stock updates on a regular basis. The application sends the messages to the broker on an MQInput node, and the message flow provides a conversion from the input currency to a number of output currencies. Include a Publication node for each currency supported, and set the *Subscription Point* to a value that reflects the currency in which the stock price is published by the node, for example, Sterling, or USD.

Configuring the Publication node

When you have put an instance of the Publication node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Publication node as follows:

1. Select the *Implicit Stream Naming* check box to take the name of the WebSphere MQ queue on which the message was received by the message flow as the stream name. This property provides forward compatibility with WebSphere MQ Publish/Subscribe, and applies to messages with an MQRFH header when MQPSSStream is not specified.

Clear the check box if you do not want this action to be taken.

2. Specify the *Subscription Point* for this Publication node. If you do not specify a value for this property, the default subscription point is assumed. This value uniquely identifies the node, and can be used by subscribers to get a specific publication (as described in the example scenario above).

For more information, refer to Subscription points.

3. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
4. Click **Apply** to make the changes to the Publication node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Publication node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Publication node Basic properties are described in the following table.

Property	M	C	Default	Description
Implicit Stream Naming	Yes	No	Cleared	Whether to take the name of the WebSphere MQ queue on which the input message was received as the stream name. If you select the check box, this action is performed.
Subscription Point	No	No		The subscription point value for the node.

The Publication node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Real-timeInput node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the Real-timeInput node”
- “Terminals and properties” on page 622

Purpose

Use the Real-timeInput node to receive messages, from clients that connect to the broker using the WebSphere MQ Real-time Transport or the WebSphere MQ Multicast Transport and that use JMS application programming interfaces, into a message flow.

The Real-timeInput node handles messages in the following message domains:

- JMSMap
- JMSStream

An output node in a message flow that starts with a Real-timeInput node can be any of the supported output nodes, including user-defined output nodes. You can create a message flow that receives messages from real-time clients and generates messages for clients that use all supported transports to connect to the broker, because you can configure the message flow to request the broker to provide any conversion that is necessary.

If you are create a message flow to use as a subflow, you cannot use a standard input node: you must use an instance of the Input node as the first node to create an in terminal for the subflow.

If your message flow does not receive messages from JMS applications, you can choose one of the supported input nodes.

The Real-timeInput node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following sample to see how you can use this node:

- Scribble sample

Configuring the Real-timeInput node

When you have put an instance of the Real-timeInput node into a message flow, you can configure it. Right-click the node in the editor view and select **Properties**. The node’s basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Real-timeInput node as follows:

1. In *Port*, identify the number of the port on which the node listens for messages from JMS applications. Ensure that the port number that you specify does not conflict with any other listener service. There is no default for this property; you must enter a value.
2. If you want to authenticate users that send messages on receipt of their messages, select the *Authentication* check box. If you clear the check box (the default setting), users are not authenticated.
3. If you want clients to use HTTP tunneling, select the *Tunnel through HTTP* check box. If you clear the check box (the default setting), messages do not use HTTP tunneling. If you set this option, all client applications that connect must use this feature. If they do not, their connection is rejected. The client application cannot use this option in conjunction with the connect-via proxy setting, which is activated from the client side.
4. In *Read Threads*, enter the number of threads that you want the broker to allocate to read messages. The broker starts as many instances of the message flow as are necessary to process current messages, up to this limit. The default setting is 10.
5. In *Write Threads*, enter the number of threads that you want the broker to allocate to write messages. The broker starts as many instances of the message flow as are necessary to process current messages, up to this limit. The default setting is 10.
6. In *Authentication Threads*, enter the number of threads that you want the broker to allocate to user authentication checks. The user authentication check is performed when a message is received. The broker starts as many instances of the message flow as are necessary to process current messages, up to this limit. The default setting is 10.
7. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
8. Click **Apply** to make the changes to the Real-timeInput node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

The Real-timeInput node routes each message that it retrieves successfully to the out terminal. If this fails, the message is retried.

Terminals and properties

The Real-timeInput node terminals are described in the following table.

Terminal	Description
Out	The output terminal to which the message is routed if it is successfully retrieved from JMS.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties

dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Real-timeInput node Basic properties are described in the following table.

Property	M	C	Default	Description
Port	Yes	Yes	0	The port number on which the input node listens for publish or subscribe requests.
Authentication	Yes	No	Cleared	Select the check box to authenticate users.
Tunnel through HTTP	Yes	No	Cleared	Select the check box to indicate that users use HTTP tunneling. Clear the check box to indicate that HTTP tunneling is not used.
Read Threads	No	Yes	10	The number of threads used for reading.
Write Threads	No	Yes	10	The number of threads used for writing.
Authentication Threads	No	Yes	10	The number of threads used for accepting connections and authenticating users.

The properties of the General Message Options for the Real-timeInput node are described in the following table.

Property	M	C	Default	Description
Parse Timing	Yes	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to “Parsing on demand” on page 706 for a full description of this property.

The Real-timeInput node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Real-timeOptimizedFlow node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 624
- “Configuring the Real-timeOptimizedFlow node” on page 624
- “Terminals and properties” on page 625

Purpose

Use the Real-timeOptimizedFlow node to receive messages from clients that connect using the WebSphere MQ Real-time Transport or the WebSphere MQ Multicast Transport, and that use JMS application programming interfaces.

The Real-timeOptimizedFlow node is a complete message flow that provides a high performance publish/subscribe message flow. The actions taken by this node are all internalized; you cannot influence its operation except by configuring its properties, and you cannot connect it to any other node.

This node also supports publication to, or subscription from, standard WebSphere MQ applications, but its performance for these applications is not as good as the performance achieved for JMS applications.

You cannot affect the message content in any way when you use the Real-timeOptimizedFlow node. If you want to modify the input message, or if you want to send messages or make publications available to applications that use other communications protocols, you must use the Real-timeInput node.

The Real-timeOptimizedFlow node is represented in the workbench by the following icon:



Using this node in a message flow

Include the Real-timeOptimizedFlow node in a message flow when you want to distribute messages through a broker to and from client applications that use JMS.

Configuring the Real-timeOptimizedFlow node

When you have put an instance of the Real-timeOptimizedFlow node into a message flow, you can configure it. Right-click the node in the editor view and select **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Real-timeOptimizedFlow node as follows:

1. In *Port*, identify the number of the port on which the node listens for publish or subscribe requests from JMS applications. Ensure that the port number that you specify does not conflict with any other listener service. There is no default for this property; you must enter a value.
2. If you want users to authenticate that send messages on receipt of their messages, select the *Authentication* check box. If you clear the check box (the default setting), users are not authenticated.
3. If you want clients to use HTTP tunneling, select the *Tunnel through HTTP* check box. If you clear the check box (the default setting), messages do not use HTTP tunneling. If you select the check box, all client applications that connect must use this feature. If they do not, their connection is rejected. The client application cannot use this option in conjunction with the connect-via proxy setting, which is activated from the client side.
4. In *Read Threads*, enter the number of threads that you want the broker to allocate to read messages. The broker starts as many instances of the message flow as are necessary to process current messages, up to this limit. The default setting is 10.

5. In *Write Threads*, enter the number of threads that you want the broker to allocate to write messages. The broker starts as many instances of the message flow as are necessary to process current messages, up to this limit. The default setting is 10.
6. In *Authentication Threads*, enter the number of threads that you want the broker to allocate to user authentication checks. The user authentication check is performed when a message is received. The broker starts as many instances of the message flow as are necessary to process current messages, up to this limit. The default setting is 10.
7. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
8. Click **Apply** to make the changes to the Real-timeOptimizedFlow node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Real-timeOptimizedFlow node has no terminals. It is a complete message flow and cannot be connected to other nodes to extend the message processing.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Real-timeOptimizedFlow node Basic properties are described in the following table.

Property	M	C	Default	Description
Port	Yes	Yes		The port number on which the node listens for publish or subscribe requests. You must provide a value for this property.
Authentication	Yes	No	Cleared	Select the check box to authenticate users.
Tunnel through HTTP	Yes	No	Cleared	Select the check box to indicate that clients use HTTP tunneling. Clear the check box to indicate that HTTP tunneling is not used.
Read Threads	No	Yes	10	The number of threads used for reading.
Write Threads	No	Yes	10	The number of threads used for writing.
Authentication Threads	No	Yes	10	The number of threads used for accepting connections and authenticating users.

The Real-timeOptimizedFlow node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

ResetContentDescriptor node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the ResetContentDescriptor node” on page 627
- “Terminals and properties” on page 628

Purpose

Use the ResetContentDescriptor node to request that the message is reparsed by a different parser. If the new parser is MRM, you can also specify a different message template (message set, type, and format). This node does not reparse the message, but the properties that you set for this node determine how the message is parsed when it is next reparsed by the message flow.

The node associates the new parser information with the input message bit stream. If the message has already been parsed to create a message tree, and the contents of the tree have been modified (for example, by a Compute node), the ResetContentDescriptor node must invoke the current parser associated with the message to parse the message and recreate the bit stream.

If your message flow has updated the message before it is received by the ResetContentDescriptor node, you must ensure that the changed message contents are still valid for the current parser. If this is not the case, the parser generates an error when it attempts to recreate the bit stream from the message tree, and the ResetContentDescriptor node raises an exception. For example, if you have added a new field to a message in the MRM domain, and the field is not present in the model, the recreation of the bit stream fails.

The ResetContentDescriptor node does **not**:

- Change the message content. It changes message properties to specify the way in which the bit stream is parsed next time that the parser is invoked.
- Convert the message from one format to another. For example, if the incoming message has a Message Format of XML and the outgoing Message Format is CWF, the ResetContentDescriptor node does not do any reformatting. It invokes the parser to recreate the bit stream of the incoming XML message, which retains the XML tags in the message. When the message is reparsed by a subsequent node, the XML tags are invalid and the reparse fails.

The ResetContentDescriptor node is represented in the workbench by the following icon:



Using this node in a message flow

For an example of how to use this node, assume that you want to swap between the BLOB and the MRM domains. The format of an incoming message might be unknown when it enters a message flow, so the BLOB parser is invoked. Later on in the message flow, you might decide that the message is predefined as a message in the MRM domain, and you can use the ResetContentDescriptor node to set the correct values to use when the message is parsed by a subsequent node in the message flow.

The following table shows typical ResetContentDescriptor node properties.

Property	Value
Message Domain	MRM
Reset Message Domain	Selected
Message Set	DH53CU406U001
Reset Message Set	Selected
Message Type	m_MESSAGE1
Reset Message Type	Selected
Message Format	CWF
Reset Message Format	Selected

The Message Domain is set to MRM, and the MRM parser is invoked when the message is next parsed. The message set, type, and format are the message template values that define the message model, and all the reset check boxes are selected because all the properties need to change.

The ResetContentDescriptor node causes the BLOB parser associated with the input message to construct the physical bit stream of the message (not the logical tree representation of it) which is later passed to the MRM parser. The MRM parser then parses the bit stream using the message template (Message Set, Message Type, and Message Format) specified in this ResetContentDescriptor node.

Configuring the ResetContentDescriptor node

When you have put an instance of the ResetContentDescriptor node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the ResetContentDescriptor node as follows:

1. If you want a different parser associated with the message, specify the new domain In *Message Domain*:
 - MRM
 - XML
 - JMSMap
 - JMSStream
 - MIME
 - BLOB

You can also specify a user-defined parser if appropriate.

Select the *Reset Message Domain* check box.

2. If the MRM parser is to reparse the message, specify the other properties of the model that are to be associated with the input message, and select the *Reset...* check box beneath each field. If the MRM parser is already associated with the input message, you have to specify only the properties that are to change.
 - a. Enter the message set in *Message Set*. Choose a value from the drop-down list of available message sets (the name and identifier of the message set are shown).

- b. Enter the identifier of the message in *Message Type*. You can find this identifier in the properties of the message in the editor view. You specified the message identifier when you created the message. Enter the identifier exactly as shown in the message properties.
- c. Enter the format of the message in *Message Format*. This specifies the wire format for the MRM parser. You can select one of the formats from the drop-down list (which lists the identifiers of those formats that you have defined on the message set specified above).

These properties set the domain, set, type, and format that you want in the message header of the message that you want to pass through the ResetContentDescriptor node. However, this only happens if suitable headers already exist. If the message does not have an MQRFH2 header, the node does not create one.

- 3. Leave *Message Set*, *Message Type*, and *Message Format* blank if you have specified the XML, JMS, MIME, or BLOB parser.
- 4. Select Validation in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

- 5. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
- 6. Click **Apply** to make the changes to the ResetContentDescriptor node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The ResetContentDescriptor node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if an error is detected by the node.
Out	The output terminal to which the message is routed if a new parser is identified by the properties.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The ResetContentDescriptor node Basic properties are described in the following table.

Property	M	C	Default	Description
Message Domain	No	No		The message domain associated with the message that you want to reparse.

Property	M	C	Default	Description
Reset Message Domain	Yes	No	Cleared	Whether to reset the message domain. If you select the check box, this action is performed.
Message Set	No	No		The message set associated with the message that you want to reparse.
Reset Message Set	Yes	No	Cleared	Whether to reset the message set. If you select the check box, this action is performed.
Message Type	No	No		The message type associated with the message that you want to reparse.
Reset Message Type	Yes	No	Cleared	Whether to reset the message type. If you select the check box, this action is performed.
Message Format	No	No		The message format associated with the message that you want to reparse.
Reset Message Format	Yes	No	Cleared	Whether to reset the message format. If you select the check box, this action is performed.

The Validation properties of the ResetContentDescriptor node are described in the following table. Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	None	Whether validation takes place. Valid values are None, Content, Content and Value, and Inherit.
Failure Action	Yes	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content and Value or Content. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	Yes	No	None	This property cannot be edited. Valid values are None, and Full.

The properties of the General Message Options for the ResetContentDescriptor node are described in the following table.

Property	M	C	Default	Description
Parse Timing	Yes	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to “Parsing on demand” on page 706 for a full description of this property.
Use MQRFH2C Compact Parser for MQRFH2 Domain	No	No	False	This property controls whether the MQRFH2C Compact Parser, instead of the MQRFH2 parser, is used for MQRFH2 headers.

The XMLNSC parser options for the ResetContentDescriptor node are described in the following table.

Property	M	C	Default	Description
Use XMLNS Compact Parser for XMLNS Domain	Yes	Cleared	No	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.
Mixed Content Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.
Comments Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.
Processing Instructions Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The ResetContentDescriptor node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

RouteToLabel node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 631
- “Configuring the RouteToLabel node” on page 631
- “Terminals and properties” on page 632

Purpose

Use the RouteToLabel node in combination with one or more Label nodes to dynamically determine the route that a message takes through the message flow, based on its content. The RouteToLabel node interrogates the LocalEnvironment of the message to determine the identifier of the Label node to which to route the message.

You must precede the RouteToLabel node in the message flow with a Compute node that populates the LocalEnvironment of the message with the identifiers of one or more Label nodes introducing the next sequence of processing for the message. The destinations are set up as a list of label names in the LocalEnvironment tree in a specific location. This excerpt of ESQL from the Airline

Reservations sample (linked to below) provides an example of how to set up the LocalEnvironment content in a Compute node:

```
IF InputRoot.XML.PassengerQuery.ReservationNumber<>' ' THEN
  SET OutputLocalEnvironment.Destination.RouterList.DestinationData[1].labelName = 'SinglePassenger';
ELSE
  SET OutputLocalEnvironment.Destination.RouterList.DestinationData[1].labelName = 'AllReservations';
END IF;
```

The label names can be any string value, and can be explicitly specified in the Compute node, taken or cast from any field in the message, or retrieved from a database. A label name in the LocalEnvironment must match the *Label Name* property of a corresponding Label Node.

When you configure the Compute node, you must also select a value for the *Compute Mode* property from the drop-down list that includes LocalEnvironment.

Design your message flow such that a RouteToLabel node logically precedes one or more Label nodes within a message flow, but do not physically wire the RouteToLabel node with a Label node. The connection is made by the broker, when required, according to the contents of LocalEnvironment.

The RouteToLabel node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following sample to see how you can use this node:

- Airline Reservations sample

Configuring the RouteToLabel node

When you have put an instance of the RouteToLabel node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the RouteToLabel node as follows:

1. Set *Mode*. This defines how the RouteToLabel node removes destinations from the LocalEnvironment associated with the message. You can set one of two values:
 - Route To First removes the first item from LocalEnvironment. The current message is routed to the Label node identified by labelName in that list item.
 - Route To Last (the default) removes the last item from LocalEnvironment. The current message is routed to the Label node identified by labelName in that list item.
2. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
3. Click **Apply** to make the changes to the RouteToLabel node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The RouteToLabel node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if a failure is detected during processing.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The RouteToLabel node Basic properties are described in the following table.

Property	M	C	Default	Description
Mode	Yes	No	Route To Last	How the RouteToLabel node processes the items within LocalEnvironment associated with the current message

The RouteToLabel node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

SCADAInput node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 634
- “Configuring the SCADAInput node” on page 634
- “Terminals and properties” on page 637

Purpose

Use the SCADAInput node to receive messages from clients that connect to the broker across the WebSphere MQ Telemetry Transport. SCADA device clients use the MQIsdp protocol to send messages, which are converted by the SCADAInput node into a format recognized by WebSphere Message Broker. The node also establishes the processing environment for these messages.

Message flows that handle messages received from SCADA devices must always start with a SCADAInput node. Set the SCADAInput node’s properties to control the way that messages are received: for example, you can indicate that a message is to be processed under transaction control.

When you deploy message flows containing SCADA nodes to a broker, you must deploy them to a single execution group, regardless of the number of message flows.

Because SCADA is primarily publish/subscribe, you typically include a Publication node to terminate the flow. In scenarios where you do not want a Publication node, you can include a SCADAOutput node. If you do so, you must also include a SCADAInput node regardless of the source of the messages, because the SCADAInput node provides the connectivity information required by the SCADAOutput node.

If you include an output node in a message flow that starts with a SCADAInput node, it can be any of the supported output nodes, including user-defined output nodes. You can create a message flow that receives messages from SCADA devices and generates messages for clients that use all supported transports to connect to the broker, because you can configure the message flow to request the broker to provide any necessary conversion.

You can request that the broker start or stop a SCADA listener by publishing messages with a specific topic. This can be done for all ports or for a single port identified in the message.

The SCADAInput node handles messages in the following message domains:

- MRM
- XML
- XMLNS
- XMLNSC
- JMSMap
- JMSStream
- MIME
- BLOB
- IDOC

You cannot use SCADAInput nodes in message flows that are to be deployed on z/OS systems.

If you want to process the data in an incoming SCADA message, include a node like the ResetContentDescriptor node and set its properties to force the bit stream to be re-parsed by a subsequent node.

If you create a message flow to use as a subflow, you cannot use a standard input node, you must use an instance of the Input node as the first node to create an in terminal for the subflow.

If your message flow does not receive messages across SCADA connections, you can choose one of the supported input nodes.

The SCADAInput node is represented in the workbench by the following icon:



Using this node in a message flow

For an example of how to use this node, assume that you create a message flow with a SCADAInput node that receives messages from a remote sensor when it detects a change in its operating environment (for example, a drop in outside temperature). You connect the node to an MQOutput node that makes these messages available on a queue serviced by a WebSphere MQ application that analyses and responds to the information received.

In a second example, you create a message flow with a SCADAInput node that receives messages each minute from a remote system. The messages contain details of the system's switch settings. The data received is fed into a ResetContentDescriptor node to cast the data from binary (BLOB) to MRM message format. The information about the system is stored in a database using the Database node, and enriched using a Compute node to create an XML message, which is published using a Publication node.

Because XML messages are expensive to send (because satellite transmission has a high cost for each byte), it is advantageous to use this method because data is enriched by the broker.

Configuring the SCADAInput node

When you have put an instance of the SCADAInput node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed in the properties dialog.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the SCADAInput node as follows:

1. Set the following basic properties:
 - a. The *Enable listener on startup* check box is initially selected. This means that the listener for MQIsdp clients is initialized when the message flow is deployed.

You can update the status of the listener by publishing on the control topic `$/SYS/SCADA/MQIsdpListener/<port_number>` with the Payload part of the message set to ON or OFF.
 - b. Specify the *Port* number on which the MQIsdp server is to listen. This must be a unique port, and must not conflict with other listeners (for example, those set up for WebSphere MQ or WebSphere MQ Everyplace). The default number is 1883.
 - c. Set the *Max Threads* value to indicate the maximum number of threads available to the MQIsdp server to support clients. The default value is 500.

If you are using DB2 for your broker database, you must specify a value that is less than or equal to the value that you have set for the DB2 configuration parameters *maxappls* and *maxagents*. See *Configuring access to databases* for further information.
 - d. Select *Use Thread Pooling* if you want the node to use a pool of threads to service clients. If you select this option, the number of threads available to the MQIsdp server is limited by *Max Threads*, which you are recommended to set to a value of between 20 and 40. If you do not select this option, a new thread is created for each client that connects. The check box is initially clear.

Use this option only if you expect a large number of clients (greater than 200) to connect.

2. Select Default in the properties dialog navigator and set values for the properties that describe the message domain, message set, message type, and message format that the node uses to determine how to parse the incoming message, and the default topic associated with the message.
 - If the incoming message has an MQRFH2 header, you do not have to set values for the Default properties because the values can be derived from the <mcd> folder in the MQRFH2 header. For example:

```
<mcd><Msd>MRM</Msd><Set>DHM4U0906S001</Set><Type>receiptmsg1</Type>
<Fmt>XML</Fmt></mcd>
```

If you set values, and those values differ from those in the MQRFH2 header, the MQRFH2 header values take precedence.
 - In *Message Domain*, select the name of the parser that you are using from the drop-down list. You can choose from the following names:
 - MRM
 - XML
 - XMLNS
 - XMLNSC
 - JMSMap
 - JMSStream
 - MIME
 - BLOB
 - IDOC
 - If you are using the MRM or IDOC parser, select the correct message set from the drop-down list in *Message Set*. This list is populated with available message sets when you select MRM or IDOC as the domain.

Leave *Message Set* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
 - If you are using the MRM parser, select the correct message from the drop-down list in *Message Type*. This list is populated with messages that are defined in the message set that you have selected.

Leave *Message Type* blank for XML, XMLNS, XMLNSC, JMS, MIME, BLOB, and IDOC parsers.
 - If you are using the MRM or IDOC parser, select the format of the message from the drop-down list in *Message Format*. This list includes all the physical formats that you have defined for this message set.

Leave *Message Format* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
3. Select Advanced in the properties dialog navigator and set the required value for *Transaction Mode* to define the transactional characteristics of how this message is handled:
 - If you select Automatic, the incoming message is received under syncpoint if it is marked persistent, otherwise it is not. The transactionality of any derived messages subsequently sent by an output node is determined by the incoming persistence property, unless the output node has explicitly overridden transactionality.
 - If you select Yes, the incoming message is received under syncpoint. Any derived messages subsequently sent by an output node in the same instance of the message flow are sent transactionally, unless the output node has explicitly overridden transactionality.

- If you select No, the incoming message is not received under syncpoint. Any derived messages subsequently sent by an output node in the flow are sent non-transactionally, unless the output node has specified that the message should be put under syncpoint.
4. Select Validation in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)
For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.
 5. Select General Message Options in the properties dialog navigator. *Parse Timing* is, by default, set to On Demand. This causes validation to be delayed until it is parsed by partial parsing. If you change this to Immediate, partial parsing is overridden and everything in the message is parsed and validated, except those complex types with a Composition of Choice or Message that cannot be resolved at the time. If you change this to Complete, partial parsing is overridden and everything in the message is parsed and validated; complex types with a Composition of Choice or Message that cannot be resolved at the time cause a validation failure.
 6. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
 7. Click **Apply** to make the changes to the SCADAInput node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Connecting the terminals:

SCADAInput routes each message that it retrieves successfully to the out terminal. If this fails, the message is propagated to the failure terminal; you can connect nodes to this terminal to handle this condition. If you have not connected the failure terminal, the message loops continually through the node until the problem is resolved.

If the message is caught by this node after an exception has been thrown further on in the message flow, the message is routed to the catch terminal. If you have not connected the catch terminal, the message loops continually through the node until the problem is resolved. Ensure that a node is always connected to this terminal if there is the possibility of the message rolling back within a message flow.

Configuring for coordinated transactions:

When you include a SCADAInput node in a message flow, the value that you set for *Transaction Mode* defines whether messages are received under syncpoint:

- If you set it to Yes (the default), the message is received under syncpoint (that is, within a WebSphere MQ unit of work). Any messages subsequently sent by an output node in the same instance of the message flow are put under syncpoint, unless the output node has explicitly overridden this.
- If you set it to Automatic, the message is received under syncpoint if the incoming message is marked persistent. Otherwise, it is not. Any message

subsequently sent by an output node is put under syncpoint, as determined by the incoming persistence property, unless the output node has explicitly overridden this.

- If you set it to No, the message is not received under syncpoint. Any messages subsequently sent by an output node in the flow are not put under syncpoint, unless an individual output node has specified that the message should be put under syncpoint.

(The MQOutput node is the only output node that you can configure to override this option.)

Terminals and properties

The SCADAInput node terminals are described in the following table.

Terminal	Description
Failure	The output terminal to which the message is routed if an error occurs.
Out	The output terminal to which the message is routed if it is successfully retrieved from the queue.
Catch	The output terminal to which the message is routed if an exception is thrown downstream and caught by this node.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The SCADAInput node Basic properties are described in the following table.

Property	M	C	Default	Description
Enable listener on startup	Yes	No	Selected	When the listener is started. If you select the check box, the listener starts when the message flow is started by the broker. If you clear the check box, the listener starts on the arrival of a message on the specified port.
Port	Yes	Yes	1883	The port on which the SCADA protocol is listening.
Max Threads	Yes	Yes	500	The maximum number of threads to be started to support SCADA devices.
Use Thread Pooling	Yes	Yes	Cleared	Whether to use thread pooling. If you select the check box, this action is performed.

The SCADAInput node Default properties are described in the following table.

Property	M	C	Default	Description
Message Domain	No	No		The domain that will be used to parse the incoming message.
Message Set	No	No		The name or identifier of the message set in which the incoming message is defined.
Message Type	No	No		The name of the incoming message.
Message Format	No	No		The name of the physical format of the incoming message.

The SCADAInput node Advanced property are described in the following table.

Property	M	C	Default	Description
Transaction Mode	Yes	No	Yes	Whether the incoming message is received under syncpoint. Valid values are Automatic, Yes, and No.

The Validation properties of the SCADAInput node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	None	Whether validation takes place. Valid values are None, Content and Value, and Content.
Failure Action	Yes	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	Yes	No	None	This property cannot be edited.

The properties of the General Message Options for the SCADAInput node are described in the following table.

Property	M	C	Default	Description
Parse Timing	Yes	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to “Parsing on demand” on page 706 for a full description of this property.
Use MQRFH2C Compact Parser for MQRFH2 Domain	No	No	False	This property controls whether the MQRFH2C Compact Parser, instead of the MQRFH2 parser, is used for MQRFH2 headers.

The properties of the XMLNSC Parser Options for the SCADAInput node are described in the following table.

Property	M	C	Default	Description
Use XMLNSC Compact Parser for XMLNS Domain	Yes	Cleared	No	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.
Mixed Content Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.

Property	M	C	Default	Description
Comments Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.
Processing Instructions Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The Description properties of the SCADAInput node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

SCADAOutput node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 640
- “Configuring the SCADAOutput node” on page 640
- “Terminals and properties” on page 641

Purpose

Use the SCADAOutput node to send a message to a client that connects to the broker using the MQIsdp protocol across the WebSphere MQ Telemetry Transport. You would typically use the Publication node to send output to a SCADA client. The SCADAOutput node lets you write your own Publication node.

If you include a SCADAOutput node in a message flow, you must also include a SCADAInput node, regardless of the source of the messages, because the SCADAInput node provides the connectivity information required by the SCADAOutput node.

When you deploy message flows containing SCADA nodes to a broker, you must deploy them to a single execution group, regardless of the number of message flows.

You cannot use the SCADAOutput node to change the transactional characteristics of the message flow. The transactional characteristics set by the message flow’s input node determine the transactional behavior of the flow.

You cannot use SCADAOutput nodes in message flows that you deploy to z/OS systems.

If you create a message flow to use as a subflow, you cannot use a standard output node; you must use an instance of the Output node to create an out terminal for the subflow through which the message can be propagated.

If you do not want your message flow to send messages to a SCADA device, you can choose another supported output node.

The SCADAOutput node is represented in the workbench by the following icon:



Using this node in a message flow

Use the Publication node to publish messages for SCADA devices. Use this node if you want to process the publication messages in a particular way for these devices.

Configuring the SCADAOutput node

When you have put an instance of the SCADAOutput node into a message flow you can configure it. Right-click the node in the editor view and click **Properties**. The node's description properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the SCADAOutput node as follows:

1. Select **Validation** in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)

For more details refer to "Validating messages" on page 79 and "Validation properties for messages in the MRM domain" on page 703.

2. Select **Description** in the properties dialog navigator to enter a short description, a long description, or both.
3. Click **Apply** to make the changes to the SCADAOutput node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Now connect the node's terminals to determine how it operates within this message flow.

Connecting the terminals:

Connect the in terminal to the node from which messages bound for SCADA destinations are routed.

Connect the out or failure terminal of this node to another node in this message flow to process the message further, process errors, or send the message to an additional destination.

If you do, the LocalEnvironment associated with the message is enhanced with the following information for each destination to which the message has been put by this node:

- Queue name
- Queue manager name
- Message reply identifier (this is set to the same value as message ID)
- Message ID (from the MQMD)
- Correlation ID (from the MQMD)

These values are written in WrittenDestination within the LocalEnvironment tree structure.

If you do not connect either terminal, the LocalEnvironment tree is unchanged.

Terminals and properties

The SCADAOutput node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if a failure is detected when the message is put to the output queue.
Out	The output terminal to which the message is routed if it has been successfully put to the output queue, and if further processing is required within this message flow.

The following table describes the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Validation properties of the SCADAOutput node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	Inherit	Whether validation takes place. Valid values are None, Content and Value, Content, and Inherit.
Failure Action	Yes	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	Yes	No	None	This property cannot be edited.

The SCADAOutput node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Throw node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the Throw node”
- “Terminals and properties” on page 643

Purpose

Use the Throw node to throw an exception within a message flow. The exception can be caught and processed by:

- A preceding TryCatch node
- The message flow input node (the built-in nodes HTTPInput, MQInput, and SCADAInput all have catch terminals)
- A preceding AggregateReply node

You can include a Throw node to force an error path through the message flow if the content of the message contains unexpected data. For example, to back out a message that does not contain a particular field, you can check (using a Filter node) that the field exists. If it does not, the message can be passed to a Throw node that records details about the exception in the ExceptionList subtree within the message.

The Throw node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following samples to see how to use this node:

- Airline Reservations sample
- Error Handler sample
- Large Messaging sample

Include a Throw node with a TryCatch node in your message flow to alert the systems administrator of a potential error situation. For example, if you have a Compute node that calculates a number, you can test the result of this calculation and Throw an exception if the result exceeds a certain amount. The TryCatch node catches this exception and propagates the message to a sequence of nodes that process the error.

Configuring the Throw node

When you have put an instance of the Throw node into a message flow, you can configure it. Right-click the node in the editor view and select **Properties**. The node’s basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Throw node as follows:

1. In *Message Catalog*, enter the fully-qualified path and file name of the message catalog that contains the message source. This can be your own message catalog, or the default message catalog supplied with WebSphere Message Broker. To use the default supplied catalog, leave this property blank.
2. In *Message Number*, enter the error number of the exception being thrown.
 If you have created your own message catalog, enter the number for the message in the catalog that you want to use when this exception is thrown.
 If you are using the default message catalog, specify a number between 3001 and 3049. These numbers are reserved in the WebSphere Message Broker catalog for your use. The text of each of these messages in the default message catalog is identical, but you can use a different number within this range for each situation in which you throw an exception; use the number to identify the exact cause of the error.
 The default message number is 3001.
3. In *Message Text*, enter any additional free format text that contains information that you want to include with the message when it is written to the local error log. For example, if you have checked for the existence of a particular field in a message and thrown an exception when that field is not found, you might include the text:
 The message did not contain the required field: Branch number

 If you are using the default message catalog, this text is inserted as &1 in the message text.
4. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
5. Click **Apply** to make the changes to the Throw node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.
 Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Throw node terminal is described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Throw node Basic properties are described in the following table.

Property	M	C	Default	Description
Message Catalog	No	No		The name of the message catalog from which the error text for the error number of the exception is extracted. The default value (blank) indicates that the message is taken from the message catalog supplied with WebSphere Message Broker.
Message Number	No	No	3001	The error number of the exception being thrown.
Message Text	No	No		Additional text that explains the cause of the error.

The Throw node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

TimeoutControl node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the TimeoutControl node” on page 645
- “Terminals and properties” on page 646

Purpose

The TimeoutControl node receives an input message that contains a timeout request. See “Timeout request message” on page 402 for a description of the timeout request message. The node validates the request, stores the message, and propagates the message (unchanged) to the next node in the message flow.

The TimeoutControl node is represented in the workbench by the following icon:



Using this node in a message flow

Use a TimeoutControl node and a TimeoutNotification node together in a message flow for an application that requires events to occur at particular times, or at regular intervals.

The following are examples of when you might want to use the timeout nodes in a message flow:

1. You need to run a batch job every day at midnight.
2. You want information about currency exchange rates to be sent to banks at hourly intervals.
3. You want to confirm that important transactions are processed within a certain time period and perform some other specified actions to warn when a transaction has not been processed in that time period.

More than one TimeoutControl node can be paired with a TimeoutNotification node. Timeout requests that are processed by those TimeoutControl nodes are all processed by the same TimeoutNotification node. This happens if the same Unique Identifier is used for the TimeoutNotification node and each of the TimeoutControl nodes.

Look at Timeout Processing sample for more details about how to use the timeout processing nodes.

Configuring the TimeoutControl node

When you have put an instance of the TimeoutControl node into a message flow, you can configure it.

Right-click the node in the editor view and click **Properties**. The node's Basic properties are displayed.

Unique Identifier is the only mandatory property. It does not have a default value.

Configure the Basic properties of the node by doing the following:

- Specify in *Unique Identifier* an identifier that is unique within the broker. This identifier should be identical to the same property in the TimeoutNotification node with which it is paired. The maximum length of this identifier is 12 characters.
- Specify, in *Request Location*, the location of the timeout request information in the incoming message. This location can be anywhere in the input message tree. If you do not specify a value for this property, `InputLocalEnvironment.TimeoutRequest` is assumed. See "Timeout request message" on page 402 for a description of the timeout request message.
- Specify a value for *Request Persistence*. This property determines whether incoming timeout requests survive a restart of either the broker or the message flow that contains the TimeoutNotification node that is paired with this TimeoutControl node. Specify **Yes** if you want the incoming request to persist; specify **No** if you do not. If you specify **Automatic**, the Persistence setting in the Properties folder of the incoming message is used. **Automatic** is the default value of this property.

Now configure the Message properties of the node:

- Specify, in *Stored Message Location*, the location of the part of the request message that you want to store for propagation by the TimeoutNotification node with which it is paired. If you do not specify a value, the entire message is stored. You can specify any valid location in the message tree. If you choose to store the entire message, you do not need to specify any values in *Message Domain*, *Message Set*, *Message Type*, or *Message Format*.
- In *Message Domain*, select the name of the parser that you are using from the drop-down list. This value, and the three corresponding values in *Message Set*, *Message Type*, and *Message Format*, are used by the TimeoutNotification node with which it is paired when it rebuilds the stored message for propagation. If you have stored the entire request message (by leaving *Stored Message Location* blank), do not specify any values here. If you choose to store part of the request message, you must specify values here that reflect the stored request message fragment as if it was the entire message, which is the case when it is processed by the TimeoutNotification node. You can choose from the following names:
 - MRM
 - XML

- XMLNS
 - XMLNSC
 - JMSMap
 - JMSStream
 - MIME
 - BLOB
 - IDOC
- If you are using the MRM or IDOC parser, select the correct message set from the drop-down list in *Message Set*.
Leave *Message Set* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.
 - If you are using the MRM parser, select the correct message from the drop-down list in *Message Type*. This list is populated with messages that are defined in the message set that you have selected.
Leave *Message Type* blank for XML, XMLNS, XMLNSC, JMS, IDOC, MIME, and BLOB parsers.
 - If you are using the MRM or IDOC parser, select the format of the message from the drop-down list in *Message Format*. This list includes all the physical formats that you have defined for this message set.
Leave *Message Format* blank for XML, XMLNS, XMLNSC, JMS, MIME, and BLOB parsers.

When you have completed your configuration, click **Apply**. This makes the changes to the TimeoutControl node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog. Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The TimeoutControl node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message tree for processing (which includes validating the timeout request specified in the message tree at <i>Request Location</i>). and adds it to the control queue.
Failure	The output terminal to which the input message is propagated if a failure is detected during processing in this node. If this terminal is not connected to another node, error information is passed back to the previous node in the message flow.
Out	The output terminal to which incoming messages are propagated, unchanged, after successful timeout request processing. If this terminal is not connected to another node, no propagation occurs. If propagation of the message fails, the message is propagated to the Failure terminal.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Basic properties of the TimeoutControl node are described in the following table.

Property	M	C	Default	Description
Unique Identifier	Yes	Yes	None	This is the only mandatory property for the node. Its value must be unique within the broker. The equivalent property of the TimeoutNotification node with which it is paired must have the same value. The maximum length of this identifier is 12 characters.
Request Location	No	No	None	This property describes where to find the timeout request information. This must be a valid location in the message tree. This is validated at runtime. If no request location is specified, LocalEnvironment.TimeoutRequest is assumed. See "Timeout request message" on page 402 for a description of the timeout request message.
Request Persistence	No	No	Automatic	This property determines whether an incoming timeout request survives a broker or message flow restart. The value of this property can be Automatic, Yes, or No. If the value is Automatic, the Persistence setting in the Properties folder of the incoming message is used.

The Message properties of the TimeoutControl node are described in the following table.

Property	M	C	Default	Description
Stored Message Location	No	No	None	The location of the part of the request message that you want to store for propagation by the TimeoutNotification node with which this node is paired.
Message Domain	No	No	None	The domain that will be used to parse the stored timeout request message by the TimeoutNotification node.
Message Set	No	No	None	The name or identifier of the message set in which the stored timeout request message is defined.
Message Type	No	No	None	The name of the stored timeout request message.
Message Format	No	No	None	The name of the physical format of the stored timeout request message.

The Description properties of the TimeoutControl node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

TimeoutNotification node

This topic contains the following sections:

- "Purpose" on page 648
- "Using this node in a message flow" on page 648
- "Configuring the TimeoutNotification node" on page 649
- "Terminals and properties" on page 650

Purpose

The TimeoutNotification node is an input node that can be used in one of two ways:

- Paired with one or more TimeoutControl nodes.

The TimeoutNotification node processes timeout request messages that are set by the TimeoutControl nodes with which it is paired, and propagates copies of the messages (or selected fragments of the messages) to the next node in the message flow.

- Standalone.

Generated messages are propagated to the next node in the message flow at time intervals that are specified in the configuration of this node.

The TimeoutNotification node is represented in the workbench by the following icon:



Using this node in a message flow

Use a TimeoutControl node and a TimeoutNotification node together in a message flow for an application that requires events to occur at a particular time, or at regular intervals.

For example, you might want a batch job to run every day at midnight, or you might want information about currency exchange rates to be sent to banks at hourly intervals.

More than one TimeoutControl node can be paired with a TimeoutNotification node. Timeout requests that are processed by those TimeoutControl nodes are all processed by the same TimeoutNotification node. This happens if the same Unique Identifier is used for the TimeoutNotification node and each of the TimeoutControl nodes.

Note that when a TimeoutNotification node is started as a result of the broker, or the message flow that contains the node, starting, it scans its internal timeout store and purges any non-persistent timeout requests. Notifications are issued for any persistent timeout requests that are now past and that have the IgnoreMissed property set to False.

If you use a TimeoutNotification node to generate a WebSphere MQ message, to an output node such as MQOutput, you must provide a valid MQMD. If the TimeoutNotification node is running in Automatic mode (standalone), this is required. If the TimeoutNotification node is running in Controlled mode (that is, it is paired with one or more TimeoutControl nodes), this is required only if the stored messages do not already have an MQMD.

The following ESQL shows how you might do this.

```
CREATE NEXTSIBLING OF OutputRoot.Properties DOMAIN 'MQMD';
SET OutputRoot.MQMD.StrucId = MQMD_STRUC_ID;
SET OutputRoot.MQMD.Version = MQMD_CURRENT_VERSION;
SET OutputRoot.MQMD.Format = 'XML';
```


Because there is no WebSphere MQ context in the local environment, the MQOutput node property *Message Context* should have the default value.

Look at Timeout Processing sample for more details about how to use the timeout processing nodes.

Configuring the TimeoutNotification node

You can configure each instance of the TimeoutNotification node in your message flow.

To configure the node, right-click the node in the editor view and click **Properties**. The node's Basic properties are displayed.

Unique Identifier is the only mandatory property. It does not have a default value.

Configure the TimeoutNotification node by doing the following:

- Specify in *Unique Identifier* a value that is unique within the broker and is the same as the identifier that is specified for the TimeoutControl nodes with which this node is paired (if there are any). The maximum length of this identifier is 12 characters.
- Specify a value for *Transaction Mode*. This property affects the transactional control of the propagated timeout messages and can be set to one of the following values:

Yes A transaction is always started.

No A transaction is never started.

Automatic

This value is only meaningful if *Operation Mode* has the value **Controlled**. Whether a transaction is started depends on the persistence of the stored timeout requests which is controlled by the value of *Request Persistence* in the TimeoutControl node with which it is paired.

- Specify a value for *Operation Mode*. This property indicates whether this node has any paired TimeoutControl nodes. Set it to one of the following values:

Automatic

The node is not paired with any TimeoutControl nodes. It generates timeout requests with an interval that is controlled by the setting of the Timeout Value property.

Controlled

The node processes all timeout requests that have been stored by the TimeoutControl nodes with which it is paired.

- If *Operation Mode* is **Automatic**, specify a value for *Timeout Interval*. This value specifies the interval (in seconds) between message propagation.
- Select **Validation** in the properties dialog navigator if you want the MRM parser to validate the body of messages against the dictionary generated from the message set. (If a message is propagated to the failure terminal of the node, it is not validated.)

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

- Select **General Message Options** in the properties dialog navigator. *Parse Timing* is, by default, set to **On Demand**. This causes validation to be delayed until it is parsed by partial parsing. If you change this to **Immediate**, partial parsing is overridden and everything in the message is parsed and validated, except those

complex types with a Composition of Choice or Message that cannot be resolved at the time. If you change this to **Complete**, partial parsing is overridden and everything in the message is parsed and validated; complex types with a Composition of Choice or Message that cannot be resolved at the time cause a validation failure.

- Select **Description** in the properties dialog navigator to enter a short description, a long description, or both.

When you have completed your configuration, click **Apply**. This makes the changes to the TimeoutNotification node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog. Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The terminals of the TimeoutNotification node are described in the following table.

Terminal	Description
Failure	The output terminal to which the message is propagated if a failure is detected during processing in this node. Nodes can be connected to this terminal to process these failures. If this terminal is not connected to another node, messages are not propagated and no logging or safe storage of data occurs.
Out	<p>The output terminal to which messages are propagated after timeouts expire.</p> <p>If the TimeoutNotification node is running in Automatic mode (that is, there are no TimeoutControl nodes paired with this node), the propagated messages contain only a Properties folder and a LocalEnvironment that is populated with the timeout information.</p> <p>If the TimeoutNotification node is running in Controlled mode (that is, TimeoutControl nodes that are paired with this node store timeout requests), the propagated messages contain what was stored by the TimeoutControl nodes; this might be entire request messages or fragments thereof.</p> <p>Note that if the TimeoutNotification node is used as the input node to a message flow that generates a WebSphere MQ message (for example, by using an MQOutput node), the message flow must create the necessary MQ headers and data (for example, MQMD).</p>
Catch	<p>The output terminal to which the message is propagated if an exception is thrown downstream. If this terminal is not connected to another node, the following events occur:</p> <ol style="list-style-type: none"> 1. The TimeoutNotification node writes the error to the local error log. 2. The TimeoutNotification node repeatedly tries to process the request until the problem that caused the exception is resolved.

The following tables describe the node properties; the column headed **M** indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed **C** indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Basic properties of the TimeoutNotification node are described in the following table.

Property	M	C	Default	Description
Unique Identifier	Yes	Yes	None	A value that is unique within the broker and that is the same as the identifier that is specified for the TimeoutControl nodes with which this node is paired (if there are any). The maximum length of this identifier is 12 characters.
Transaction Mode	No	No	'Yes'	The transaction mode for the node. Its value can be Yes, No, or Automatic. If the transaction mode is Automatic, a transaction based on the persistence of the stored messages which is controlled by the <i>Request Persistence</i> property of the TimeoutControl node with which it is paired.
Operation Mode	No	No	'Automatic'	This property indicates whether or not this node is paired with any paired TimeoutControl nodes. Its value can be Automatic or Controlled.
Timeout Interval	No	No	1	The interval (in seconds) between timeout requests. It is relevant only if <i>Operation Mode</i> is set to Automatic.

The Validation properties of the TimeoutNotification node are described in the following table.

Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	None	Whether validation takes place. Valid values are None, Content, and Content And Value.
Failure Action	Yes	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	Yes	No	None	This property cannot be edited.

The properties of the General Message Options for the TimeoutNotification node are described in the following table.

Property	M	C	Default	Description
Parse Timing	Yes	No	On Demand	This property controls when an input message is parsed. Valid values are On Demand, Immediate, and Complete. Refer to “Parsing on demand” on page 706 for a full description of this property.
Use MQRFH2C Compact Parser for MQRFH2 Domain	No	No	False	This property controls whether the MQRFH2C Compact Parser, instead of the MQRFH2 parser, is used for MQRFH2 headers.

The properties of the XMLNSC Parser Options for the TimeoutNotification node are described in the following table.

Property	M	C	Default	Description
Use XMLNSC Compact Parser for XMLNS Domain	Yes	Cleared	No	This property controls whether the XMLNSC parser is used to create elements in the message tree when it encounters messages in the XMLNS Domain.
Mixed Content Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters mixed text in an input message. Valid values are None and All. Selecting All means that elements are created for mixed text. Selecting None means that mixed text is ignored and no elements are created.
Comments Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters comments in an input message. Valid values are None and All. Selecting All means that elements are created for comments. Selecting None means that comments are ignored and no elements are created.
Processing Instructions Retain Mode	Yes	No	None	This property controls whether the XMLNSC parser creates elements in the message tree when it encounters processing instructions in an input message. Valid values are None and All. Selecting All means that elements are created for processing instructions. Selecting None means that processing instructions are ignored and no elements are created.

The Description properties of the TimeoutNotification node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Trace node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 653
- “Configuring the Trace node” on page 653
- “Terminals and properties” on page 655

Purpose

Use the Trace node to generate trace records that can incorporate text, message content, and date and time information, to help you to monitor the behavior of the message flow.

You can write the records to the user trace file, another file, or the local error log (which contains error and information messages written by all other WebSphere Message Broker components). If you write traces to the local error log, you can issue a message from the default message catalog supplied with WebSphere Message Broker, or you can create your own message catalog.

The operation of the Trace node is independent of the setting of user tracing for the message flow in which it resides. In particular, records written by the Trace node to the user trace log are written even if user trace is not currently active for the message flow.

The Trace node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following samples to see how you can use this node:

- Airline Reservations sample
- Aggregation sample
- Timeout Processing sample

Include a Trace node to help you to diagnose errors in your message flow. By tracing the contents of the message at various points in the flow, you can determine the sequence of processing. You can also configure the Trace node to record the content of a message, and to check the action of a specific node on the message. For example, you can include a Trace node immediately after a Compute node to check that the output message has the expected format.

Remove Trace nodes from your message flow when you have tested the message flow and have proved that its operation is correct.

You can also use the Trace node to provide information in error handling within your message flows. For example, you can use this node to record failures in processing due to errors in the content or format of a message.

Configuring the Trace node

When you have put an instance of the Trace node into a message flow you can configure it. Right-click the node in the editor view and select **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Trace node as follows:

1. Set the *Destination* property to indicate where to write the output from the Trace node:

- To write the trace record to the local system error log, select Local Error Log.

The information you include in the trace record is written to:

- On Windows, the Event log (Application View).
- On UNIX, the syslog.
- On z/OS, the operator console.

If you select this option, you must indicate the number of the trace message to be written, and the message catalog in which the message is defined.

- If you leave *Message Catalog* blank, the default message catalog supplied with WebSphere Message Broker is used as the source of the message to be written.

You must also enter the error number of the record in *Message Number*. Numbers 3051 to 3099 are reserved in the WebSphere Message Broker (default) catalog for this use. The text of each of these messages in the default message catalog is identical, but if you use a different number within this range for each situation that you trace, you can identify the exact cause of the error. The default message number is 3051.

- If you create your own message catalog, enter the fully-qualified file name for your catalog in *Message Catalog*.

You must also enter the appropriate number for the message in the catalog that you want to write to the local error log in *Message Number*. On some platforms, message numbers ending 00 are reserved for system use; do not include messages with numbers like 3100 in your message catalog.

- If you want to write the trace record to the system-generated user trace log, select *User Trace*.

These records are written regardless of the setting of the *User Trace* property for the deployed message flow.

The user trace is written to the `\log` subdirectory of the root directory (for example, the default on Windows 2000 is `c:\Program Files\IBM\WebSphere Message Broker`). The file name is made up of the broker name, the broker UUID, and a suffix of `userTrace.bin` (for example, `broker.e51906cb-dd00-0000-0080-b10e69a5d551.userTrace.bin.0`). Use the `mqsireadlog` and `mqsiformatlog` commands before you view the user trace log.

- If you want to write the trace record to a file of your choice, select *File*.

If you select this option, you must also set *File Path* to the fully-qualified path name for the trace. If you do not set the path, the location of the file depends on the system. For example, on z/OS, the file is created within the home directory of the broker service ID.

You can use any name for the trace file. For example, `c:\$user\trace\trace.log`.

If you specify a file that does not already exist, the file is created. However, directories are not created by this process, so the full path must already exist.

- If you don't want to write any trace records, select *None*.

2. In *Pattern*, create an ESQL pattern to specify what information to write. If you write the trace record to the local error log, the pattern governs the information that is written in the text of the message number selected. If you use the default message catalog, and a number between 3051 and 3099, the pattern information is inserted as `&1` in the message text.

- You can write plain text, which is copied into the trace record exactly as you have entered it.
- You can identify parts of the message to write to the trace record, specifying the full field identifiers enclosed within the characters `${` and `}`. To record the entire message, specify `${Root}`.
- You can use the ESQL functions to provide additional information. For example, you can use the ESQL function `CURRENT_DATE` to record the date, time, or both, at which the trace record is written.

The pattern below illustrates some of the options available. It writes an initial line of text, records two elements of the current message, and adds a simple timestamp:

Message passed through with the following fields:
 Store name is \${Body.storedetailselement.storename}
 Total sales are \${Body.totalselement.totalsales}
 Time is: \${EXTRACT(HOUR FROM CURRENT_TIMESTAMP)}
 :\${EXTRACT(MINUTE FROM CURRENT_TIMESTAMP)}

The resulting trace record is:

Message passed through with the following fields:
 Store name is 'SRUCorporation'
 Total sales are '34.98'
 Time is: 11:19

A pattern containing syntax errors does not prevent a message flow containing the Trace node from deploying but the node writes no trace records.

3. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
4. Click **Apply** to make the changes to the Trace node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The terminals of the Trace node are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Out	The output terminal through which the message is propagated.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Basic properties of the Trace node are described in the following table.

Property	M	C	Default	Description
Destination	Yes	No	User Trace	The destination of the trace record written by the node. Valid choices are User Trace, File, Local Error Log, and None.
File Path	No	Yes		The fully-qualified file name of the file to which to write records. Valid only if <i>Destination</i> is set to File.
Pattern	No	No		The data that is to be included in the trace record.
Message Catalog	No	No		The name of the message catalog from which the error text for the error number of the exception is extracted. The default value (blank) indicates that the message is taken from the message catalog supplied with WebSphere Message Broker.
Message Number	No	No	3051	The error number of the message that is written.

The Description properties of the Trace node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

TryCatch node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the TryCatch node”
- “Terminals and properties” on page 657

Purpose

Use the TryCatch node to provide a special handler for exception processing. Initially the input message is routed on the try terminal of this node, which you must connect to the remaining non-error processing nodes of the message flow. If a downstream node (which can be a Throw node) throws an exception, the TryCatch node catches it and routes the original message to its catch terminal. Connect the catch terminal to further nodes to provide error processing for the message after an exception.

In this way, exceptions are stopping the message flow processing, and from affecting any transaction in progress. If the catch output terminal is connected, the message is propagated to it. If the catch terminal is not connected, the message is discarded.

The TryCatch node is represented in the workbench by the following icon:



Using this node in a message flow

Look at the following sample to see how you can use this node:

- Error Handler sample

Use the Throw and TryCatch nodes when you are use the Compute node to calculate a total. You can create a message that is sent to your system administrator when the total calculated exceeds the maximum value for the Total field.

Configuring the TryCatch node

When you have put an instance of the TryCatch node into a message flow, you can configure it.

To do this, right-click the node in the editor view and click **Properties**. The Description properties are displayed.

Enter a short description, a long description, or both.

Click **Apply** to make the changes to the TryCatch node without closing the properties dialog, or click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Now you must connect the node's terminals to determine how it operates within this message flow.

Connecting the terminals:

The TryCatch node has no configurable properties that affect its operation. You determine how it operates by connecting the output terminals to subsequent nodes in your message flow.

1. Connect the try terminal to the first node in the sequence of nodes that provides the normal (non-error) phase of processing of this message. This can be a sequence of one or more nodes that perform any valid processing. It can conclude with an output node, but does not have to.
2. Connect the catch terminal to the first node in the sequence of nodes that provides the error processing for this message flow. This can be a sequence of one or more nodes that perform any valid processing. It can conclude with an output node, but does not have to.

When an exception is thrown in the message flow, either by the explicit use of the Throw node or the ESQL THROW statement, or by the broker raising an implicit exception when it detects an error that the message flow is not programmed to handle, control returns to the TryCatch node.

The message is propagated through the catch terminal and the error handling that you have designed is executed. The message that is propagated through this terminal has the content that it had at the point at which the exception was thrown, including the full description of the exception in the ExceptionList.

Terminals and properties

The TryCatch node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Catch	The output terminal to which the message is propagated if an exception is thrown downstream and caught by this node.
Try	The output terminal to which the message is propagated if it is not caught.

The following table describes the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The TryCatch node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Validate node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 659
- “Configuring the Validate node” on page 659
- “Terminals and properties” on page 660

Purpose

Use the Validate node to check that the message that arrives on its input terminal is as expected.

You can check that the message has the expected message template properties (that is, the message domain, message set and message type).

You can also check that the content of the message is correct by selecting message validation.

The checks that can be performed depend on the domain of the message. See the following table:

Check	Domain
Check message domain	All domains
Check message set	MRM and IDOC only
Check message type	MRM only
Validate message body	MRM and IDOC only

You can check the message against one or more of message domain, message set or message type. The property is checked only if you select its corresponding checkbox, which means that a property containing an empty string can be compared.

You can check the content of the message by giving a value to the Validate property. Validation takes place if the *Validate* property is set to a value other than None, which is the default value.

For validation failures to be returned to the Validate node from the parser, the *Failure Action* property must be set to either Exception or Exception List. Otherwise, validation failures are simply logged.

If all the specified checks pass, the message is propagated through the Match terminal of the node.

If any of the checks fail, the message is propagated through the Failure terminal. If the Failure terminal is not connected to some failure handling processing, an exception is thrown.

Note: The Validate node replaces the Check node which is deprecated in WebSphere Message Broker Version 6.0 and subsequent releases. The Validate node works in the same way as the Check node, but it has additional Validation properties to allow the validation of message content by parsers that support that capability.

The Validate node is represented in the workbench by the following icon:



Using this node in a message flow

You can use the Validate node to confirm that a message has the correct message template properties, and has valid content, before allowing the message into the rest of the flow. This means that subsequent nodes can rely upon the message being correct without doing their own error checking.

You can also use the Validate node to ensure that the message is routed appropriately through the message flow. For example, you can configure it to direct a message that requests stock purchases through a different route from that required for a message that requests stock sales.

Another routing example is the receipt of electronic messages from your staff at your head office. These messages are used for multiple purposes, for example to request technical support or stationery, or to advise you about new customer leads. These messages can be processed automatically because your staff fill in a standard form. If you want these messages to be processed separately from other messages received, use the Validate node to ensure that only staff messages that have a specific message type are processed by this message flow.

Configuring the Validate node

When you have put an instance of the Validate node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Validate node as follows:

1. To check the parser to be used for the incoming message, select the box *Check Domain* and choose one of the following values from the drop-down list of the *Domain* property:
 - MRM
 - XML
 - XMLNS
 - XMLNSC
 - JMSMap
 - JMSStream
 - MIME
 - BLOB
 - IDOC
2. If you are using the MRM or IDOC parser, to check that the incoming message belongs to a particular message set, select the check box *Check Set* and choose one of the values from the drop-down list of the *Set* property. This list is populated when you choose MRM or IDOC as the message domain. Leave *Set* clear for the XML, JMS, MIME and BLOB parsers.

- If you are using the MRM parser, to check that the incoming message is a particular message type, select the check box *Check Type* and enter the name of the message in the *Type* property.

Leave *Type* clear unless you are using the MRM parser. Note that the IDOC parser automatically obtains the message type from the message.

- If you are using the MRM or IDOC parser, to validate the body of messages against the dictionary generated from the message set, select the desired validation properties from the Validation properties dialog.

For more details refer to “Validating messages” on page 79 and “Validation properties for messages in the MRM domain” on page 703.

- Select Description in the properties dialog navigator to enter a short description, a long description, or both.
- Click **Apply** to make the changes to the Validate node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The terminals of the Validate node are described in the following table.

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the message is routed if the incoming message does not match the specified properties.
Match	The output terminal to which the message is routed if the incoming message matches the specified properties.

The following tables describe the properties of the Validate node; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Basic properties of the Validate node are described in the following table.

Property	M	C	Default	Description
Domain	No	No		The name of the domain.
Check Domain	Yes	No	Cleared	Whether to check the incoming message against the Domain property. If you select the check box, this action is performed.
Set	No	No		The name or identifier of the message set to which the incoming message belongs.
Check Set	Yes	No	Cleared	Whether to check the incoming message against the Set property. If you select the check box, this action is performed.
Type	No	No		The message identifier.
Check Type	Yes	No	Cleared	Whether to check the incoming message against the Type property. If you select the check box, this action is performed.

The Validation properties of the Validate node are described in the following table. Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description of these properties.

Property	M	C	Default	Description
Validate	Yes	Yes	None	Whether validation takes place. Valid values are None, Content and Value, and Content, and Inherit.
Failure Action	Yes	No	Exception	What happens if validation fails. You can set this property only if you set <i>Validate</i> to Content or Content and Value. Valid values are User Trace, Local Error Log, Exception, and Exception List.
Include All Value Constraints	Yes	No	Selected	This property cannot be edited. The default action, indicated by the check box being selected, is that basic value constraint checks are included in Content and Value validation.
Fix	Yes	No	None	This property cannot be edited. Valid values are None, and Full.

The Description properties of the Validate node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

Warehouse node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow” on page 662
- “Configuring the Warehouse node” on page 662
- “Terminals and properties” on page 664

Purpose

Use the Warehouse node to interact with a database in the specified ODBC data source. The Warehouse node is a specialized form of the Database node that stores the entire message, or parts of the message, or both, in a table within the database. You define what is stored by defining mappings that use the data from the input message to identify the action required.

You can use the message warehouse:

- To maintain an audit trail of messages passing through the broker.
- For offline or batch processing of messages that have passed through the broker (data mining).
- As a source from which to reprocess selected messages in the broker.

You can retrieve messages that you have stored in the warehouse using standard database query and mining techniques. No explicit support is provided by WebSphere Message Broker.

You must have created (or identified, if someone else created them for you):

- Input data in the form of a message set and message.
- An ODBC connection to the database.
- A database and database table to store the message.
- At least two columns in the table: one for the binary object (the message), one for the timestamp.

The Warehouse node is represented in the workbench by the following icon:



Using this node in a message flow

When you use the Warehouse node, you can choose to store in the database associated with the node:

- The entire message, optionally with an associated timestamp. The message is stored as a binary object, with the timestamp in a separate column. There are two advantages to this:
 1. You do not have to decide beforehand how you will use the warehoused data; because you have stored it all, you can retrieve all the data and apply data mining tools to it later.
 2. You do not have to define a specific database schema for every type of message that might pass through the broker. In a complex system, there might be many different message types and the overhead of defining a unique schema for each message type can become prohibitive. You can precede each Warehouse node with a Compute node that converts each message into a canonical warehouse format with a common schema, or you can store the whole message as a binary object.
- Selected parts of the message, optionally with an associated timestamp. Doing this requires a defined database schema for that message type. The message is mapped to true type so, for example, a character string in the message is stored as a character string in the database.

Configuring the Warehouse node

When you have put an instance of the Warehouse node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the Warehouse node as follows:

1. Specify in *Data Source* the name by which the appropriate database is known on the system on which this message flow is to execute. The broker connects to this database with user ID and password information that you have specified on the `mqsicreatebroker`, `mqsichangebroker`, or `mqsisetdbparms` command. On z/OS systems, the broker uses the broker started task ID, or the user ID and password that were specified on the `mqsisetdbparms` command JCL, BIPSDBP in the customization data set <hlq>.SBIPPROC.
2. In *Field Mapping*, identify the mapping routine that is to be executed in this node. By default, the name assigned to the mapping routine is identical to the name of the mappings file in which the routine is defined, and the default

name for the file is the name of the message flow concatenated with the name of the node when you include it in the message flow (for example, MFlow1_Warehouse.mfmap for the first Warehouse node in message flow MFlow1). You cannot specify a value that includes spaces.

If you click **Browse** next to this entry field, a dialog is displayed that lists all available mapping routines that are accessible by this node. Select the routine that you want and click **OK**. The routine name is set in *Field Mapping*.

To work with the mapping routine associated with this node, right-click the node and select **Open Mappings**. If the mapping routine does not exist, it is created for you with the default name in the default file. If the file already exists, you can also open file <flow_name>_<node_name>.mfmap in the Navigator view.

The content of the mapping routine determines what is stored in the database, and in what format. You can choose, for example, to store all or just a part of each message. You can also choose to store the data as binary data, or to store each field in the same format as it is in the message (for example, a character field in the message is stored as character in the database).

A mapping routine is specific to the type of node with which it is associated; you cannot use a mapping routine that you have developed for a Warehouse node with any other node that uses mappings (for example, a DataInsert node). If you create a mapping routine, you cannot call it from any other mapping routine, although you can call it from an ESQL routine.

3. Select the *Transaction* setting from the drop-down menu. The values are:
 - **Automatic** (the default). The message flow, of which the Warehouse node is a part, is committed if it is successful. That is, the actions that you define in the mappings are taken and the message continues through the message flow. If the message flow fails, it is rolled back. Therefore choosing **Automatic** means that the ability to commit or roll back the action of the Warehouse node on the database depends on the success or failure of the entire message flow.
 - **Commit**. If you want to commit any uncommitted actions taken in this message flow on the database connected to this node, irrespective of the success or failure of the message flow as a whole, select **Commit**. The changes to the database are committed even if the message flow itself fails.
4. Select **Basic** in the properties dialog navigator and select or clear the two check boxes:
 - If you want database warning messages to be treated as errors, and the node to propagate the output message to the failure terminal, select the *Treat Warnings as Errors* check box. The box is initially cleared.

When you select the box, the node handles all positive return codes from the database as errors and generates exceptions in the same way as it does for the negative, or more serious, errors.

If you do not select the box, the node treats warnings as normal return codes, and does not raise any exceptions. The most significant warning raised is *not found*, which can be handled as a normal return code safely in most circumstances.

- If you want the broker to generate an exception when a database error is detected, select the *Throw Exception on Database Errors* check box. The box is initially selected.

If you clear the box, you must handle the error in the message flow to ensure the integrity of the broker and the database: the error is ignored if you do not handle it through your own processing, because you have chosen not to

invoke the default error handling by the broker. For example, you could connect the failure terminal to an error processing subroutine.

5. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
6. Click **Apply** to make the changes to the Warehouse node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

Terminals and properties

The Terminals of the Warehouse node are described in the following table:

Terminal	Description
In	The input terminal that accepts a message for processing by the node.
Failure	The output terminal to which the input message is propagated if a failure is detected during the computation. If you have selected <i>Treat Warnings as Errors</i> , the node propagates the message to this terminal even if the processing completes successfully.
Out	The output terminal that outputs the message following the execution of the database statement.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The Warehouse node Basic properties are described in the following table.

Property	M	C	Default	Description
Data Source	No	Yes		The ODBC data source name of the database that contains the tables to which you refer in the mappings associated with this node (identified by the <i>Field Mapping</i> property).
Field Mapping	Yes	No	Warehouse	The name of the mapping routine that contains the statements that are to be executed against the database or the message tree. The routine is unique to this type of node.
Transaction	Yes	No	Automatic	The transaction mode for the node. This can be Automatic or Commit.
Treat Warnings as Errors	Yes	No	Cleared	Treat database SQL warnings as errors. If you select the check box, the action is performed.
Throw Exception on Database Error	Yes	No	Selected	Database errors cause the broker to throw an exception. If you select the check box, the action is performed.

The Description properties of the Warehouse node are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.

Property	M	C	Default	Description
Long Description	No	No		Text that describes the purpose of the node in the message flow.

XMLTransformation node

This topic contains the following sections:

- “Purpose”
- “Using this node in a message flow”
- “Configuring the XMLTransformation node” on page 666
- “Terminals and properties” on page 668

Purpose

Use the XMLTransformation node to transform an XML message to another form of XML message, according to the rules provided by an XSL (eXtensible Stylesheet Language) style sheet.

You can specify the location of the style sheet to apply to this transformation in one of three ways:

1. You can use node properties. This ensures that the transformation defined by this single style sheet is applied to every message processed by this node.
2. You can use the content of the XML data within the message itself. This transforms the message according to a style sheet that the message itself defines. This behavior is only available for XSL and XML files that are located within a Message Flow project.
3. You can set a value within the LocalEnvironment folder associated with the message. This provides a dynamic choice of style sheet, because you must set this value (in a Compute node) within the message flow after receipt of the message. You can therefore use a variety of inputs to determine which style sheet to use for this message, such as the content of the message data or a value in a database.

An XSLT (eXtensible Stylesheet Language for Transformations) compiler is used for the transformation if the style sheet is not embedded within the message and the node cache level (node property Stylesheet Cache Level) is greater than 0.

The XMLTransformation node is represented in the workbench by the following icon:



Using this node in a message flow

For an example of how to use this node, consider two news organizations that exchange information on a regular basis. One might be a television station, the other a newspaper. Although the information is similar, the vocabulary used by the two is different. This node can transform one format to the other by applying the rules of the specified style sheet. If you specify the style sheet in the message (either the XML data or the LocalEnvironment), the same node can perform both transformations.

Look at XMLT sample for more details about how to use the XMLTransformation node.

Configuring the XMLTransformation node

When you have put an instance of the XMLTransformation node into a message flow, you can configure it. Right-click the node in the editor view and click **Properties**. The node's basic properties are displayed.

All mandatory properties for which you must enter a value (those that do not have a default value defined) are marked with an asterisk on the properties dialog.

Configure the XMLTransformation node as follows:

1. Select values for the *XML Embedded Selection Priority*, *Message Environment Selection Priority*, and *Broker Node Attribute Selection Priority* properties. The values that you set determine the order in which alternative locations are searched for the stylesheet information. The highest priority setting is 1. The default order is:
 - a. *XML Embedded Selection Priority*, which therefore has a default value of 1. The node searches the XML data for stylesheet location information. For example, the XML data might contain:

```
<?xml-stylesheet type="text/xsl" href="foo.xsl"?>
```
 - b. *Message Environment Selection Priority*, which therefore has a default value of 2. The node searches the LocalEnvironment associated with the current message for the stylesheet information stored in an element called `ComIbmXslXsltStylesheetname`.
Because this node was available in a SupportPac for Version 2.1, and element `ComIbmXslMqsiStylesheetname` was used for the name of the style sheet, the current node checks both elements. If both are present, the value in `ComIbmXslXsltStylesheetname` takes precedence.
 - c. *Broker Node Attribute Selection Priority*, which therefore has a default value of 3. The node uses the node properties *Stylesheet Name* and *Stylesheet Directory* to determine the correct values.

You can set more than one property to the same value, although this is not recommended. If you do, the order of priority set by the node is the default order indicated above.

If you set a value of ignore, the node does not search the corresponding location for the stylesheet identification. If you set all three properties to ignore, a runtime error is generated.

2. If you want to specify a non-deployed style sheet using node properties, enter the required value for *Stylesheet Name*. This value is ignored if stylesheet information is searched for and found in a preferred location (determined by the selection priority values that you have set).

If you want to specify a principal style sheet, there are two ways of doing this:

- a. Using the **Browse** button near the *Stylesheet Name* property field from the workspace. The identified principal style sheet and all its relatively referenced descendant style sheets are automatically added to the bar file when adding a message flow to a .bar file (as long as both they and their parent style sheets are available).
- b. For the identification of an already deployed or to be deployed style sheet, only the *Stylesheet Name* property can be used, and the *Stylesheet Directory* property must be left empty.

3. If the stylesheet identification is fully qualified, *Stylesheet Directory* is ignored; if it is not, the value that you set in this property is added to the beginning of the specification, regardless of where it is found.
4. In *Stylesheet Cache Level*, specify the number of compiled or parsed stylesheets that are stored in this instance of the node. The default value is 5. The stylesheet cache is retained for the life of the node. It is cleared when the node is deleted from the flow, or when the flow is deleted, or when the execution group is stopped. If you modify a style sheet, the modified (latest) version is used in preference to the cached version. If you want to refresh the cache, use the `mqsireload` command.
5. Select *Output Character Set* in the properties dialog navigator to specify the order in which the node searches valid locations to find the character set to use for the output message. The highest priority setting is 1. The default order is:
 - a. *Message Environment Selection Priority*, which therefore has a default value of 1. The node searches the LocalEnvironment associated with the current message for the character set information stored in an element called `ComIbmXslXsltOutputcharset`.
 For example, to encode the output of the transformation as UTF-8, enter the value `1208` as a string in this element.
 Because this node was available in a SupportPac for Version 2.1, and element `ComIbmXslMqsiOutputcharset` was used for the output character set, the current node checks both elements. If both are present, the value in `ComIbmXslXsltOutputcharset` takes precedence.
 - b. *Broker Node Attribute Selection Priority*, which therefore has a default value of 2. The node uses the property *Output Character Set* to determine the correct value.
 If you set a value for *Output Character Set*, the value that you enter must be numeric. For example, to encode the output of the transformation as UTF-16, enter `1200`.

You can set more than one property to the same value, although this is not recommended. If you do, the order of priority set by the node is the default order indicated above.

If you set a value of `0`, the node does not search the corresponding location for the character set identification.

If the node cannot determine the output character set from either of these two sources, either because no value is set or because the selection priorities are set to `0`, the default value `1208` (UTF8) is used. (The XSL specification indicates that the output character set can be specified in the style sheet; however, the XMLTransformation node ignores this value.)

6. Select *Detail Trace* in the properties dialog navigator to trace the actions of the XMLTransformation node. The default value for the *Detail Trace* property is `0ff`. To activate trace, set the property to `0n`.

The trace information is written to a trace file `XMLTTrace.log`:

- On z/OS systems, the file is located in `<broker_dir>/output`, where `<broker_dir>` is the directory in which you have installed the broker
- On Windows systems, the file is located in `<broker work path>\common\log`.
- On UNIX systems, the file is located in `<broker work path>\common\log`.

If you set detailed trace on for one XMLTransformation node, it is on for all nodes in the execution group.

Note: This property is now deprecated. Any relevant trace now goes into the user trace, providing that user debug trace is enabled. The setting of *Detail Trace* in the XMLTransformation node does not affect any user trace.

7. Select Description in the properties dialog navigator to enter a short description, a long description, or both.
8. Click **Apply** to make the changes to the XMLTransformation node without closing the properties dialog. Click **OK** to apply the changes and close the properties dialog.

Click **Cancel** to close the dialog and discard all the changes that you have made to the properties.

If you are dealing with large XML messages and receive an 'out of memory' error, you can use the **mqsireportproperties** command to see the current value of the Java Heap size for the XSLT engine, and the **mqsichangeproperties** command to increase it:

```
mqsireportproperties brokerName -e executionGroupLabel
                                -o ComIbmJVMMManager -n jvmMaxHeapSize
mqsichangeproperties brokerName -e executionGroupLabel
                                -o ComIbmJVMMManager -n jvmMaxHeapSize -v newSize
```

replacing brokerName, executionGroupLabel, and newSize with the appropriate values.

The value that you should choose for newSize is dependent upon the amount of physical memory that your computer has and how much you are using Java. A value in the range 512 MB to 1 GB is suggested.

Terminals and properties

The XMLTransformation node terminals are described in the following table.

Terminal	Description
In	The input terminal that accepts the message for processing by the node.
Failure	The output terminal to which the original message is routed if an error is detected during transformation.
Out	The output terminal to which the successfully transformed message is routed.

The following tables describe the node properties; the column headed M indicates whether the property is *mandatory* (marked with an asterisk on the properties dialog if you must enter a value when no default is defined), the column headed C indicates whether the property is *configurable* (you can change the value when you add the message flow to the bar file to deploy it).

The XMLTransformation node Stylesheet properties are described in the following table.

Property	M	C	Default	Description
XML Embedded Selection Priority	Yes	No	1	The priority value for searching for stylesheet location information in the XML data.
Message Environment Selection Priority	Yes	No	2	The priority value for searching for stylesheet location information in the LocalEnvironment folder of the current message.

Property	M	C	Default	Description
Broker Node Attribute Selection Priority	Yes	No	3	The priority value for searching for stylesheet location information as a property of the node
Stylesheet Name	No	Yes		The name of the style sheet, used if the stylesheet specification is searched for in node properties.
Stylesheet Directory	No	Yes		The path where the style sheet is located. Used by all location methods.
Stylesheet Cache Level	No	No	5	The number of compiled or parsed stylesheets that are stored in this instance of the node.

The XMLTransformation node Output Character Set properties are described in the following table.

Property	M	C	Default	Description
Message Environment Selection Priority	Yes	No	1	The priority value for searching for the Output Character Set ID in the LocalEnvironment folder of the current message.
Broker Node Attribute Selection Priority	Yes	No	2	The priority value for searching for the Output Character Set ID as a property of the node.
Output Character Set	No	No		The numeric value of the Output Character Set

The XMLTransformation node Detail Trace properties are described in the following table.

Property	M	C	Default	Description
Trace Setting	Yes	No	Off	Whether tracing is on or off. If tracing is on, low level tracing is recorded in a file.

The XMLTransformation node Description properties are described in the following table.

Property	M	C	Default	Description
Short Description	No	No		A brief description of the node.
Long Description	No	No		Text that describes the purpose of the node in the message flow.

If the prologue of the input message body contains an XML encoding declaration, The XMLTransformation node ignores the encoding, and always uses the CodedCharSetId in the message property folder to decode the message.

Deployment of deployed style sheets or XML files

In order to use deployed style sheets or XML files, you must do the following:

1. **Make sure the files have the correct file name extensions:** Style sheets to be deployed must have either .xsl or .xslt as their file extension and XML files to be deployed must have .xml as their file extension.
2. **Import the files into the Eclipse workspace:** All style sheets and XML files that are to be deployed must be imported into an Eclipse workspace project. Location dependent descendant style sheets or XML files that are to be deployed must be placed in the correct directory structure relative to their

parent style sheets. You should not put in the Eclipse workspace Location dependent descendants that you do not want to deploy.

3. **Make sure all references to the files are relative:** Generally speaking, all references to a principal style sheet that can appear in a flow's local environment, in an input XML document, in the XMLTransformation node *Stylesheet Name* property, or in a parent style sheet, must be made relative. A reference to a principal style sheet should be made relative to the root of the relevant Eclipse workspace project. For example, a reference to a principal style sheet in the Eclipse workspace, `C:\project1\A\b.xml` must be specified as `a/b.xml` (or `./a/b.xml`). The only exception is, when you specify a principal style sheet as the *Stylesheet Name* property on an XMLTransformation node, you can also use an absolute path that points to the correct directory structure in the Eclipse workspace. If the principal style sheet is found, the system automatically resets the node property to the correct relative value (and performs an automatic deployment of the principal style sheet, together with all its location dependent descendant style sheets that are available in the relevant Eclipse workspace project). All references to the location dependent descendant style sheets (or XML files) of a principal style sheet must be made relative to the location of their parent style sheets. For example, if style sheet, `//project1/a/b.xml` references style sheet, `//project1/a/c/d.xml` the reference must be changed to `c/d.xml` (or `./c/d.xml`).
4. **Deal with non-deployed child style sheets or XML files:** If you have a relatively referenced child style sheet (or XML file) that is not to be deployed yet its parent is, make sure it is placed in the right place under `/XSL/external` (`/XML/external`). A broker automatically associates the execution group deployed storage tree, `/XSL/external`, and `/XML/external` tree, together. That means, for example, that the broker automatically performs a search in `/XML/external/a/b` directory for a reference of document (`b/c.xml`) in the deployed principal style sheet `a/style.xml` if `b/c.xml` is not found in the broker's deployed storage. Relative path references must also be used for files that are known to have been deployed but which are not available in the workspace.
5. **Deploy the files:** You only need to manually deploy style sheets or XML files that are not picked up by the system (the tooling provides warnings about these files). To manually deploy, add the files to be deployed to a broker archive (refer to "Adding files to a broker archive" on page 432 and to "Adding keywords to XSL stylesheets") and deploy the broker archive. You need only deploy a style sheet or XML file once for all message flows that belong to the same execution group and at least once for each execution group. It is not necessary to deploy style sheets or XML files in the same `.bar` file which contains the flows referencing them. Also, style sheets and XML files in a `.bar` file do not have to be referenced by any flows. However, when a flow is saved to a `.bar` file, warnings are given if any child style sheets or XML files are not found in the `.bar` file. It is your responsibility to ensure that these child style sheets or XML files are available on the broker. A broker is not Eclipse-project aware. Therefore, a deployed style sheet that belongs to one project can be overwritten by another if they share the same directory path and file name, even if they belonged to different projects. A deployment overwrites, without warning, all existing old versions of the files that are being deployed. This applies to automatically deployed style sheets as well.

Adding keywords to XSL stylesheets

Keywords can be embedded at any place in an XSL stylesheet. The keyword can be added as an XML comment and must have the following format:

```
$MQSI keyword = value MQSI$
```

The example shows how to add a keyword of *author* with the value *John* to an XML stylesheet:

```
<?xml version="1.0" encoding="UTF-8">
<!-- $MQSI author = John MQSI$ >
<xsl:stylesheet version="1.0"
xmlns:xsl=http://www.w3.org/1999/XSL/Transform>
<xsl:output method="text" encoding="UTF-8"/>
<xsl:template match="/">
<xsl:value-of select="message"/>
</xsl:template>
</xsl:stylesheet>
```

The Configuration Manager does not extract `version="1.0"` from this example. This is because the value is not bounded by the `$MQSI` and `MQSI$` keywords.

User-defined nodes

You can define your own nodes to use in WebSphere Message Broker message flows, to add to the function provided by the supplied or built-in nodes. You can also use nodes created and supplied by independent software vendors and other companies.

If you, and other vendors, follow the instructions for providing help information for user-defined nodes, the associated help appears following this topic in the help when you install the user-defined node.

Supported code pages

Application messages must conform to supported code pages.

The message flows that you create, configure, and deploy to a broker can process and construct application messages in any code page listed in the table. You can also generate a new code page converter.

This behavior might be affected by the use of other products with WebSphere Message Broker. Check the documentation for other products, including any databases that you use, for further code page support information.

If you experience code page translation problems on HP-UX, check the WebSphere MQ queue manager attribute *CodedCharSetID* (CCSID). The default value for this attribute is 1051. Change this to 819 for queue managers that host WebSphere Message Broker components.

For detailed information about Chinese code page GB18030 support, see “Chinese code page GB18030” on page 699.

WebSphere Message Broker supports the codepages given in the following tables by default. To find a code page for a specific CCSID, search for an internal converter name in the form *ibm-ccsid*, where *ccsid* is the CCSID you are looking for.

- Unicode converters
- European and American language converters
- Asian language converters
- Windows US and European converters
- MAC related converters

- Hebrew, Cyrillic and ECMA language converters
- Indian language converters
- EBCDIC converters

Table 4. Unicode converters

Internal converter name	Aliases
UTF-8	UTF-8 ibm-1208 ibm-1209 ibm-5304 ibm-5305 windows-65001 cp1208
UTF-16	UTF-16 ISO-10646-UCS-2 unicode csUnicode ucs-2
UTF-16BE	UTF-16BE x-utf-16be ibm-1200 ibm-1201 ibm-5297 ibm-13488 ibm-17584 windows-1201 cp1200 cp1201 UTF16_BigEndian
UTF-16LE	UTF-16LE x-utf-16le ibm-1202 ibm-13490 ibm-17586 UTF16_LittleEndian windows-1200
UTF-32	UTF-32 ISO-10646-UCS-4 csUCS4 ucs-4
UTF-32BE	UTF-32BE UTF32_BigEndian ibm-1232 ibm-1233

Table 4. Unicode converters (continued)

Internal converter name	Aliases
UTF-32LE	UTF-32LE UTF32_LittleEndian ibm-1234
UTF16_PlatformEndian	UTF16_PlatformEndian
UTF16_OppositeEndian	UTF16_OppositeEndian
UTF32_PlatformEndian	UTF32_PlatformEndian
UTF32_OppositeEndian	UTF32_OppositeEndian
UTF-7	UTF-7 windows-65000
IMAP-mailbox-name	IMAP-mailbox-name
SCSU	SCSU
BOCU-1	BOCU-1 csBOCU-1
CESU-8	CESU-8

Table 5. European and American language converters

Internal converter name	Aliases
ISO-8859-1	ISO-8859-1 ibm-819 IBM819 cp819 latin1 8859_1 csISOLatin1 iso-ir-100 ISO_8859-1:1987 I1 819
US-ASCII	US-ASCII ASCII ANSI_X3.4-1968 ANSI_X3.4-1986 ISO_646.irv:1991 iso_646.irv:1983 ISO646-US us csASCII iso-ir-6 cp367 ascii7 646 windows-20127

Table 5. European and American language converters (continued)

Internal converter name	Aliases
gb18030	gb18030 ibm-1392 windows-54936
ibm-367_P100-1995	ibm-367_P100-1995 ibm-367 IBM367
ibm-912_P100-1995	ibm-912_P100-1995 ibm-912 iso-8859-2 ISO_8859-2:1987 latin2 csISOLatin2 iso-ir-101 l2 8859_2 cp912 912 windows-28592
ibm-913_P100-2000	ibm-913_P100-2000 ibm-913 iso-8859-3 ISO_8859-3:1988 latin3 csISOLatin3 iso-ir-109 l3 8859_3 cp913 913 windows-28593
ibm-914_P100-1995	ibm-914_P100-1995 ibm-914 iso-8859-4 latin4 csISOLatin4 iso-ir-110 ISO_8859-4:1988 l4 8859_4 cp914 914 windows-28594

Table 5. European and American language converters (continued)

Internal converter name	Aliases
ibm-915_P100-1995	ibm-915_P100-1995 ibm-915 iso-8859-5 cyrillic csISOLatinCyrillic iso-ir-144 ISO_8859-5:1988 8859_5 cp915 915 windows-28595
ibm-1089_P100-1995	ibm-1089_P100-1995 ibm-1089 iso-8859-6 arabic csISOLatinArabic iso-ir-127 ISO_8859-6:1987 ECMA-114 ASMO-708 8859_6 cp1089 1089 windows-28596 ISO-8859-6-I ISO-8859-6-E
ibm-813_P100-1995	ibm-813_P100-1995 ibm-813 iso-8859-7 greek greek8 ELOT_928 ECMA-118 csISOLatinGreek iso-ir-126 ISO_8859-7:1987 8859_7 cp813 813 windows-28597

Table 5. European and American language converters (continued)

Internal converter name	Aliases
ibm-916_P100-1995	ibm-916_P100-1995 ibm-916 iso-8859-8 hebrew csISOLatinHebrew iso-ir-138 ISO_8859-8:1988 ISO-8859-8-I ISO-8859-8-E 8859_8 cp916 916 windows-28598
ibm-920_P100-1995	ibm-920_P100-1995 ibm-920 iso-8859-9 latin5 csISOLatin5 iso-ir-148 ISO_8859-9:1989 I5 8859_9 cp920 920 windows-28599 ECMA-128
ibm-921_P100-1995	ibm-921_P100-1995 ibm-921 iso-8859-13 8859_13 cp921 921
ibm-923_P100-1998	ibm-923_P100-1998 ibm-923 iso-8859-15 Latin-9 I9 8859_15 latin0 csisolatin0 csisolatin9 iso8859_15_fdis cp923 923 windows-28605

Table 6. Asian language converters

Internal converter name	Aliases
ibm-942_P12A-1999	ibm-942_P12A-1999 ibm-942 ibm-932 cp932 shift_jis78 sjis78 ibm-942_VSUB_VPUA ibm-932_VSUB_VPUA
ibm-943_P15A-2003	ibm-943_P15A-2003 ibm-943 Shift_JIS MS_Kanji csShiftJIS windows-31j csWindows31J x-sjis x-ms-cp932 cp932 windows-932 cp943c IBM-943C ms932 pck sjis ibm-943_VSUB_VPUA
ibm-943_P130-1999	ibm-943_P130-1999 ibm-943 Shift_JIS cp943 943 ibm-943_VASCII_VSUB_VPUA
ibm-33722_P12A-1999	ibm-33722_P12A-1999 ibm-33722 ibm-5050 EUC-JP Extended_UNIX_Code_Packed_Format_for_Japanese csEUCPkFmtJapanese X-EUC-JP eucjis windows-51932 ibm-33722_VPUA IBM-eucJP

Table 6. Asian language converters (continued)

Internal converter name	Aliases
ibm-33722_P120-1999	ibm-33722_P120-1999 ibm-33722 ibm-5050 cp33722 33722 ibm-33722_VASCII_VPUA
ibm-954_P101-2000	ibm-954_P101-2000 ibm-954 EUC-JP
ibm-1373_P100-2002	ibm-1373_P100-2002 ibm-1373 windows-950
windows-950-2000	windows-950-2000 Big5 csBig5 windows-950 x-big5
ibm-950_P110-1999	ibm-950_P110-1999 ibm-950 cp950 950
macos-2566-10.2	macos-2566-10.2 Big5-HKSCS big5hk HKSCS-BIG5
ibm-1375_P100-2003	ibm-1375_P100-2003 ibm-1375 Big5-HKSCS
ibm-1386_P100-2002	ibm-1386_P100-2002 ibm-1386 cp1386 windows-936 ibm-1386_VSUB_VPUA
windows-936-2000	windows-936-2000 GBK CP936 MS936 windows-936

Table 6. Asian language converters (continued)

Internal converter name	Aliases
ibm-1383_P110-1999	ibm-1383_P110-1999 ibm-1383 GB2312 csGB2312 EUC-CN ibm-eucCN hp15CN cp1383 1383 ibm-1383_VPUA
ibm-5478_P100-1995	ibm-5478_P100-1995 ibm-5478 GB_2312-80 chinese iso-ir-58 csISO58GB231280 gb2312-1980 GB2312.1980-0
ibm-964_P110-1999	ibm-964_P110-1999 ibm-964 EUC-TW ibm-eucTW cns11643 cp964 964 ibm-964_VPUA
ibm-949_P110-1999	ibm-949_P110-1999 ibm-949 cp949 949 ibm-949_VASCII_VSUB_VPUA
ibm-949_P11A-1999	ibm-949_P11A-1999 ibm-949 cp949c ibm-949_VSUB_VPUA

Table 6. Asian language converters (continued)

Internal converter name	Aliases
ibm-970_P110-1995	ibm-970_P110-1995 ibm-970 EUC-KR KS_C_5601-1987 windows-51949 csEUCKR ibm-eucKR KSC_5601 5601 ibm-970_VPUA
ibm-971_P100-1995	ibm-971_P100-1995 ibm-971 ibm-971_VPUA
ibm-1363_P11B-1998	ibm-1363_P11B-1998 ibm-1363 KS_C_5601-1987 KS_C_5601-1989 KSC_5601 csKSC56011987 korean iso-ir-149 5601 cp1363 ksc windows-949 ibm-1363_VSUB_VPUA
ibm-1363_P110-1997	ibm-1363_P110-1997 ibm-1363 ibm-1363_VASCII_VSUB_VPUA
windows-949-2000	windows-949-2000 windows-949 KS_C_5601-1987 KS_C_5601-1989 KSC_5601 csKSC56011987 korean iso-ir-149 ms949
ibm-1162_P100-1999	ibm-1162_P100-1999 ibm-1162

Table 6. Asian language converters (continued)

Internal converter name	Aliases
ibm-874_P100-1995	ibm-874_P100-1995 ibm-874 ibm-9066 cp874 TIS-620 tis620.2533 eucTH cp9066
windows-874-2000	windows-874-2000 TIS-620 windows-874 MS874

Table 7. Windows US and European converters

Internal converter name	Aliases
ibm-437_P100-1995	ibm-437_P100-1995 ibm-437 IBM437 cp437 437 csPC8CodePage437 windows-437
ibm-850_P100-1995	ibm-850_P100-1995 ibm-850 IBM850 cp850 850 csPC850Multilingual windows-850
ibm-851_P100-1995	ibm-851_P100-1995 ibm-851 IBM851 cp851 851 csPC851
ibm-852_P100-1995	ibm-852_P100-1995 ibm-852 IBM852 cp852 852 csPCp852 windows-852

Table 7. Windows US and European converters (continued)

Internal converter name	Aliases
ibm-855_P100-1995	ibm-855_P100-1995 ibm-855 IBM855 cp855 855 csIBM855 csPCp855
ibm-856_P100-1995	ibm-856_P100-1995 ibm-856 cp856 856
ibm-857_P100-1995	ibm-857_P100-1995 ibm-857 IBM857 cp857 857 csIBM857 windows-857
ibm-858_P100-1997	ibm-858_P100-1997 ibm-858 IBM00858 CCSID00858 CP00858 PC-Multilingual-850+euro cp858
ibm-860_P100-1995	ibm-860_P100-1995 ibm-860 IBM860 cp860 860 csIBM860
ibm-861_P100-1995	ibm-861_P100-1995 ibm-861 IBM861 cp861 861 cp-is csIBM861 windows-861

Table 7. Windows US and European converters (continued)

Internal converter name	Aliases
ibm-862_P100-1995	ibm-862_P100-1995 ibm-862 IBM862 cp862 862 csPC862LatinHebrew DOS-862 windows-862
ibm-863_P100-1995	ibm-863_P100-1995 ibm-863 IBM863 cp863 863 csIBM863
ibm-864_X110-1999	ibm-864_X110-1999 ibm-864 IBM864 cp864 csIBM864
ibm-865_P100-1995	ibm-865_P100-1995 ibm-865 IBM865 cp865 865 csIBM865
ibm-866_P100-1995	ibm-866_P100-1995 ibm-866 IBM866 cp866 866 csIBM866 windows-866
ibm-867_P100-1998	ibm-867_P100-1998 ibm-867 cp867
ibm-868_P100-1995	ibm-868_P100-1995 ibm-868 IBM868 cp868 868 csIBM868 cp-ar

Table 7. Windows US and European converters (continued)

Internal converter name	Aliases
ibm-869_P100-1995	ibm-869_P100-1995 ibm-869 IBM869 cp869 869 cp-gr csIBM869 windows-869
ibm-878_P100-1996	ibm-878_P100-1996 ibm-878 KOI8-R koi8 csKOI8R cp878
ibm-901_P100-1999	ibm-901_P100-1999 ibm-901_P100-1999 ibm-901
ibm-902_P100-1999	ibm-902_P100-1999 ibm-902
ibm-922_P100-1999	ibm-922_P100-1999 ibm-922 cp922 922
ibm-4909_P100-1999	ibm-4909_P100-1999 ibm-4909
ibm-5346_P100-1998	ibm-5346_P100-1998 ibm-5346 windows-1250 cp1250
ibm-5347_P100-1998	ibm-5347_P100-1998 ibm-5347 windows-1251 cp1251
ibm-5348_P100-1997	ibm-5348_P100-1997 ibm-5348 windows-1252 cp1252
ibm-5349_P100-1998	ibm-5349_P100-1998 ibm-5349 windows-1253 cp1253

Table 7. Windows US and European converters (continued)

Internal converter name	Aliases
ibm-5350_P100-1998	ibm-5350_P100-1998 ibm-5350 windows-1254 cp1254
ibm-9447_P100-2002	ibm-9447_P100-2002 ibm-9447 windows-1255 cp1255
windows-1256-2000	windows-1256-2000 windows-1256 cp1256
ibm-9449_P100-2002	ibm-9449_P100-2002 ibm-9449 windows-1257 cp1257
ibm-5354_P100-1998	ibm-5354_P100-1998 ibm-5354 windows-1258 cp1258
ibm-1250_P100-1995	ibm-1250_P100-1995 ibm-1250 windows-1250
ibm-1251_P100-1995	ibm-1251_P100-1995 ibm-1251 windows-1251
ibm-1252_P100-2000	ibm-1252_P100-2000 ibm-1252 windows-1252
ibm-1253_P100-1995	ibm-1253_P100-1995 ibm-1253 windows-1253
ibm-1254_P100-1995	ibm-1254_P100-1995 ibm-1254 windows-1254
ibm-1255_P100-1995	ibm-1255_P100-1995 ibm-1255
ibm-5351_P100-1998	ibm-5351_P100-1998 ibm-5351 windows-1255
ibm-1256_P110-1997	ibm-1256_P110-1997 ibm-1256

Table 7. Windows US and European converters (continued)

Internal converter name	Aliases
ibm-5352_P100-1998	ibm-5352_P100-1998 ibm-5352 windows-1256
ibm-1257_P100-1995	ibm-1257_P100-1995 ibm-1257
ibm-5353_P100-1998	ibm-5353_P100-1998 ibm-5353 windows-1257
ibm-1258_P100-1997	ibm-1258_P100-1997 ibm-1258 windows-1258

Table 8. MAC related converters

Internal converter name	Aliases
macos-0_2-10.2	macos-0_2-10.2 macintosh mac csMacintosh windows-10000
macos-6-10.2	macos-6-10.2 x-mac-greek windows-10006 macgr
macos-7_3-10.2	macos-7_3-10.2 x-mac-cyrillic windows-10007 maccy
macos-29-10.2	macos-29-10.2 x-mac-centraleurroman windows-10029 x-mac-ce macce
macos-35-10.2	macos-35-10.2 x-mac-turkish windows-10081 mactr
ibm-1051_P100-1995	ibm-1051_P100-1995 ibm-1051 hp-roman8 roman8 r8 csHPRoman8

Table 8. MAC related converters (continued)

Internal converter name	Aliases
ibm-1276_P100-1995	ibm-1276_P100-1995 ibm-1276 Adobe-Standard-Encoding csAdobeStandardEncoding
ibm-1277_P100-1995	ibm-1277_P100-1995 ibm-1277 Adobe-Latin1-Encoding

Table 9. Hebrew, Cyrillic, and ECMA language converters

Internal converter name	Aliases
ibm-1006_P100-1995	ibm-1006_P100-1995 ibm-1006 cp1006 1006
ibm-1098_P100-1995	ibm-1098_P100-1995 ibm-1098 cp1098 1098
ibm-1124_P100-1996	ibm-1124_P100-1996 ibm-1124 cp1124 1124
ibm-1125_P100-1997	ibm-1125_P100-1997 ibm-1125 cp1125
ibm-1129_P100-1997	ibm-1129_P100-1997 ibm-1129
ibm-1131_P100-1997	ibm-1131_P100-1997 ibm-1131 cp1131
ibm-1133_P100-1997	ibm-1133_P100-1997 ibm-1133
ibm-1381_P110-1999	ibm-1381_P110-1999 ibm-1381 cp1381 1381
ISO_2022,locale=ja,version=0	ISO_2022,locale=ja,version=0 ISO-2022-JP csISO2022JP

Table 9. Hebrew, Cyrillic, and ECMA language converters (continued)

Internal converter name	Aliases
ISO_2022,locale=ja,version=1	ISO_2022,locale=ja,version=1 ISO-2022-JP-1 JIS JIS_Encoding
ISO_2022,locale=ja,version=2	ISO_2022,locale=ja,version=2 ISO-2022-JP-2 csISO2022JP2
ISO_2022,locale=ja,version=3	ISO_2022,locale=ja,version=3 JIS7 csJISEncoding
ISO_2022,locale=ja,version=4	ISO_2022,locale=ja,version=4 JIS8
ISO_2022,locale=ko,version=0	ISO_2022,locale=ko,version=0 ISO-2022-KR csISO2022KR
ISO_2022,locale=ko,version=1	ISO_2022,locale=ko,version=1 ibm-25546
ISO_2022,locale=zh,version=0	ISO_2022,locale=zh,version=0 ISO-2022-CN
ISO_2022,locale=zh,version=1	ISO_2022,locale=zh,version=1 ISO-2022-CN-EXT
HZ	HZ HZ-GB-2312
ibm-897_P100-1995	ibm-897_P100-1995 ibm-897 JIS_X0201 X0201 csHalfWidthKatakana

Table 10. Indian language converters

Internal converter name	Aliases
ISCII,version=0 ISCII,version=0	x-iscii-de windows-57002 iscii-dev
ISCII,version=1 ISCII,version=1	x-iscii-be windows-57003 iscii-bng windows-57006 x-iscii-as

Table 10. Indian language converters (continued)

Internal converter name	Aliases
ISCII,version=2 ISCII,version=2	x-iscii-pa windows-57011 iscii-gur
ISCII,version=3 ISCII,version=3	x-iscii-gu windows-57010 iscii-guj
ISCII,version=4 ISCII,version=4	x-iscii-or windows-57007 iscii-ori
ISCII,version=5 ISCII,version=5	x-iscii-ta windows-57004 iscii-tml
ISCII,version=6 ISCII,version=6	x-iscii-te windows-57005 iscii-tlg
ISCII,version=7 ISCII,version=7	x-iscii-ka windows-57008 iscii-knd
ISCII,version=8 ISCII,version=8	x-iscii-ma windows-57009 iscii-mlm

Table 11. EBCDIC converters

Internal converter name	Aliases
LMBCS-1	LMBCS-1 lmbcs
LMBCS-2	LMBCS-2
LMBCS-3	LMBCS-3
LMBCS-4	LMBCS-4
LMBCS-5	LMBCS-5
LMBCS-6	LMBCS-6
LMBCS-8	LMBCS-8
LMBCS-11	LMBCS-11
LMBCS-16	LMBCS-16
LMBCS-17	LMBCS-17
LMBCS-18	LMBCS-18
LMBCS-19	LMBCS-19

Table 11. EBCDIC converters (continued)

Internal converter name	Aliases
ibm-37_P100-1995	ibm-37_P100-1995 ibm-37 IBM037 ibm-037 ebcdic-cp-us ebcdic-cp-ca ebcdic-cp-wt ebcdic-cp-nl csIBM037 cp037 037 cpibm37 cp37
ibm-273_P100-1995	ibm-273_P100-1995 ibm-273 IBM273 CP273 csIBM273 ebcdic-de cpibm273 273
ibm-277_P100-1995	ibm-277_P100-1995 ibm-277 IBM277 cp277 EBCDIC-CP-DK EBCDIC-CP-NO csIBM277 ebcdic-dk cpibm277 277
ibm-278_P100-1995	ibm-278_P100-1995 ibm-278 IBM278 cp278 ebcdic-cp-fi ebcdic-cp-se csIBM278 ebcdic-sv cpibm278 278

Table 11. EBCDIC converters (continued)

Internal converter name	Aliases
ibm-280_P100-1995	ibm-280_P100-1995 ibm-280 IBM280 CP280 ebcdic-cp-it csIBM280 cpibm280 280
ibm-284_P100-1995	ibm-284_P100-1995 ibm-284 IBM284 CP284 ebcdic-cp-es csIBM284 cpibm284 284
ibm-285_P100-1995	ibm-285_P100-1995 ibm-285 IBM285 CP285 ebcdic-cp-gb csIBM285 ebcdic-gb cpibm285 285
ibm-290_P100-1995	ibm-290_P100-1995 ibm-290 IBM290 cp290 EBCDIC-JP-kana csIBM290
ibm-297_P100-1995	ibm-297_P100-1995 ibm-297 IBM297 cp297 ebcdic-cp-fr csIBM297 cpibm297 297

Table 11. EBCDIC converters (continued)

Internal converter name	Aliases
ibm-420_X120-1999	ibm-420_X120-1999 IBM420 cp420 ebcdic-cp-ar1 csIBM420 420
ibm-424_P100-1995	ibm-424_P100-1995 ibm-424 IBM424 cp424 ebcdic-cp-he csIBM424 424
ibm-500_P100-1995	ibm-500_P100-1995 ibm-500 IBM500 CP500 ebcdic-cp-be csIBM500 ebcdic-cp-ch cpibm500 500
ibm-803_P100-1999	ibm-803_P100-1999 ibm-803 cp803
ibm-838_P100-1995	ibm-838_P100-1995 ibm-838 IBM-Thai csIBMThai cp838 838 ibm-9030
ibm-870_P100-1995	ibm-870_P100-1995 ibm-870 IBM870 CP870 ebcdic-cp-roece ebcdic-cp-yu csIBM870

Table 11. EBCDIC converters (continued)

Internal converter name	Aliases
ibm-871_P100-1995	ibm-871_P100-1995 ibm-871 IBM871 ebcdic-cp-is csIBM871 CP871 ebcdic-is cpibm871 871
ibm-875_P100-1995	ibm-875_P100-1995 ibm-875 IBM875 cp875 875
ibm-918_P100-1995	ibm-918_P100-1995 ibm-918 IBM918 CP918 ebcdic-cp-ar2 csIBM918
ibm-930_P120-1999	ibm-930_P120-1999 ibm-930 ibm-5026 cp930 cpibm930 930
ibm-933_P110-1995	ibm-933_P110-1995 ibm-933 cp933 cpibm933 933
ibm-935_P110-1999	ibm-935_P110-1999 ibm-935 cp935 cpibm935 935
ibm-937_P110-1999	ibm-937_P110-1999 ibm-937 cp937 cpibm937 937

Table 11. EBCDIC converters (continued)

Internal converter name	Aliases
ibm-939_P120-1999	ibm-939_P120-1999 ibm-939 ibm-931 ibm-5035 cp939 939
ibm-1025_P100-1995	ibm-1025_P100-1995 ibm-1025 cp1025 1025
ibm-1026_P100-1995	ibm-1026_P100-1995 ibm-1026 IBM1026 CP1026 csIBM1026 1026
ibm-1047_P100-1995	ibm-1047_P100-1995 ibm-1047 IBM1047 cpibm1047
ibm-1097_P100-1995	ibm-1097_P100-1995 ibm-1097 cp1097 1097
ibm-1112_P100-1995	ibm-1112_P100-1995 ibm-1112 cp1112 1112
ibm-1122_P100-1999	ibm-1122_P100-1999 ibm-1122 cp1122 1122
ibm-1123_P100-1995	ibm-1123_P100-1995 ibm-1123 cp1123 1123 cpibm1123
ibm-1130_P100-1997	ibm-1130_P100-1997 ibm-1130
ibm-1132_P100-1998	ibm-1132_P100-1998 ibm-1132

Table 11. EBCDIC converters (continued)

Internal converter name	Aliases
ibm-1140_P100-1997	ibm-1140_P100-1997 ibm-1140 IBM01140 CCSID01140 CP01140 cp1140 cpibm1140 ebcdic-us-37+euro
ibm-1141_P100-1997	ibm-1141_P100-1997 ibm-1141 IBM01141 CCSID01141 CP01141 cp1141 cpibm1141 ebcdic-de-273+euro
ibm-1142_P100-1997	ibm-1142_P100-1997 ibm-1142 IBM01142 CCSID01142 CP01142 cp1142 cpibm1142 ebcdic-dk-277+euro ebcdic-no-277+euro
ibm-1143_P100-1997	ibm-1143_P100-1997 ibm-1143 IBM01143 CCSID01143 CP01143 cp1143 cpibm1143 ebcdic-fi-278+euro ebcdic-se-278+euro
ibm-1144_P100-1997	ibm-1144_P100-1997 ibm-1144 IBM01144 CCSID01144 CP01144 cp1144 cpibm1144 ebcdic-it-280+euro

Table 11. EBCDIC converters (continued)

Internal converter name	Aliases
ibm-1145_P100-1997	ibm-1145_P100-1997 ibm-1145 IBM01145 CCSID01145 CP01145 cp1145 cpibm1145 ebcdic-es-284+euro
ibm-1146_P100-1997	ibm-1146_P100-1997 ibm-1146 IBM01146 CCSID01146 CP01146 cp1146 cpibm1146 ebcdic-gb-285+euro
ibm-1147_P100-1997	ibm-1147_P100-1997 ibm-1147 IBM01147 CCSID01147 CP01147 cp1147 cpibm1147 ebcdic-fr-297+euro
ibm-1148_P100-1997	ibm-1148_P100-1997 ibm-1148 IBM01148 CCSID01148 CP01148 cp1148 cpibm1148 ebcdic-international-500+euro
ibm-1149_P100-1997	ibm-1149_P100-1997 ibm-1149 IBM01149 CCSID01149 CP01149 cp1149 cpibm1149 ebcdic-is-871+euro
ibm-1153_P100-1999	ibm-1153_P100-1999 ibm-1153 cpibm1153

Table 11. EBCDIC converters (continued)

Internal converter name	Aliases
ibm-1154_P100-1999	ibm-1154_P100-1999 ibm-1154 cpibm1154
ibm-1155_P100-1999	ibm-1155_P100-1999 ibm-1155 cpibm1155
ibm-1156_P100-1999	ibm-1156_P100-1999 ibm-1156 cpibm1156
ibm-1157_P100-1999	ibm-1157_P100-1999 ibm-1157 cpibm1157
ibm-1158_P100-1999	ibm-1158_P100-1999 ibm-1158 cpibm1158
ibm-1160_P100-1999	ibm-1160_P100-1999 ibm-1160 cpibm1160
ibm-1164_P100-1999	ibm-1164_P100-1999 ibm-1164 cpibm1164
ibm-1364_P110-1997	ibm-1364_P110-1997 ibm-1364 cp1364
ibm-1371_P100-1999	ibm-1371_P100-1999 ibm-1371 cpibm1371
ibm-1388_P103-2001	ibm-1388_P103-2001 ibm-1388 ibm-9580
ibm-1390_P110-2003	ibm-1390_P110-2003 ibm-1390 cpibm1390
ibm-1399_P110-2003	ibm-1399_P110-2003 ibm-1399
ibm-16684_P110-2003	ibm-16684_P110-2003 ibm-16684
ibm-4899_P100-1998	ibm-4899_P100-1998 ibm-4899 cpibm4899

Table 11. EBCDIC converters (continued)

Internal converter name	Aliases
ibm-4971_P100-1999	ibm-4971_P100-1999 ibm-4971 cpibm4971
ibm-12712_P100-1998	ibm-12712_P100-1998 ibm-12712 cpibm12712 ebcdic-he
ibm-16804_X110-1999	ibm-16804_X110-1999 ibm-16804 cpibm16804 ebcdic-ar
ibm-1137_P100-1999	ibm-1137_P100-1999 ibm-1137
ibm-5123_P100-1999	ibm-5123_P100-1999 ibm-5123
ibm-8482_P100-1999	ibm-8482_P100-1999 ibm-8482
ibm-37_P100-1995,swaplfnl	ibm-37_P100-1995,swaplfnl ibm-37-s390 ibm037-s390
ibm-1047_P100-1995,swaplfnl	ibm-1047_P100-1995,swaplfnl ibm-1047-s390
ibm-1140_P100-1997,swaplfnl	ibm-1140_P100-1997,swaplfnl ibm-1140-s390
ibm-1142_P100-1997,swaplfnl	ibm-1142_P100-1997,swaplfnl ibm-1142-s390
ibm-1143_P100-1997,swaplfnl	ibm-1143_P100-1997,swaplfnl ibm-1143-s390
ibm-1144_P100-1997,swaplfnl	ibm-1144_P100-1997,swaplfnl ibm-1144-s390
ibm-1145_P100-1997,swaplfnl	ibm-1145_P100-1997,swaplfnl ibm-1145-s390
ibm-1146_P100-1997,swaplfnl	ibm-1146_P100-1997,swaplfnl ibm-1146-s390
ibm-1147_P100-1997,swaplfnl	ibm-1147_P100-1997,swaplfnl ibm-1147-s390
ibm-1148_P100-1997,swaplfnl	ibm-1148_P100-1997,swaplfnl ibm-1148-s390

Table 11. EBCDIC converters (continued)

Internal converter name	Aliases
ibm-1149_P100-1997,swaplfnl	ibm-1149_P100-1997,swaplfnl ibm-1149-s390
ibm-1153_P100-1999,swaplfnl	ibm-1153_P100-1999,swaplfnl ibm-1153-s390
ibm-12712_P100-1998,swaplfnl	ibm-12712_P100-1998,swaplfnl ibm-12712-s390
ibm-16804_X110-1999,swaplfnl	ibm-16804_X110-1999,swaplfnl ibm-16804-s390
ebcdic-xml-us	ebcdic-xml-us

Chinese code page GB18030

There are some restrictions if you are working with messages in Chinese code page GB18030.

The broker can input, manipulate, and output application messages encoded in code page IBM-5488 (GB18030 support) with the following restrictions:

- If you configure a message flow to store GB18030 data in character form in a user database, ensure that the database manager that you are using supports GB18030.
- To enable support for GB18030 in the workbench and Configuration Manager:
 - If you run a workbench or Configuration Manager that requires GB18030 support on a Windows 2000 system, apply the GB18030 patch supplied by Microsoft. This support is included in Windows XP.
 - Create the Configuration Manager configuration repository with a code set of UTF-8.
 - Change the text font preference in the workbench to use GB18030:
 - Select **Preferences** in the **Window** menu.
 - Expand the **Workbench** item on the left side of the Preferences dialog (click the plus sign) and select **Fonts**.
 - In the Fonts display, select Text Font. Click **Change**, and select the correct values in the Fonts selection dialog.
 - Click **OK** to confirm the selection and close the dialog.
 - Click **Apply** to apply the change, then **OK** to close the Preference dialog.

WebSphere MQ connections

The number of WebSphere MQ connections a broker requires to its queue manager depends on the actions of the message flows that access the MQ resource. For each broker flow that accesses a queue, one connection is required for every message flow thread. If a different node on the same thread uses the same queue manager, the same connection is used.

The number of queue handles required also depends on the behavior of the flow. For each flow that accesses queues, one queue handle is required for each unique queue name for every message flow thread. Nodes accessing the same queue name in the same flow use the same queue handle.

When you start a broker, and while it is running, it opens WebSphere MQ queue handles. The broker caches these queue handles. For example, when a message flow node initiates access to the first MQ resource it uses, it opens a connection for the queue manager and opens the queue. This is done the first time that a message is processed by that message flow node. For MQInput nodes this occurs when the flow is started. This queue handle remains open until:

- The message flow becomes idle and has not been used for one minute

- The execution group is stopped

- The broker is stopped

The queue handle for the input node is not released when the flow is idle. The queue handle is released only when you stop the message flow.

A thread performing WebSphere MQ work becomes idle when it has not received any messages on its input queue for one minute. The allowed idle time starts from when the input queue being read becomes empty. If a message flow gets a message from the input queue, the timer is reset.

When a message flow is idle, the execution group periodically releases WebSphere MQ queue handles. Therefore, connections held by the broker reflect the broker's current use of these resources.

User database connections

This topic describes how to determine the number of database connections a broker requires for capacity and resource planning. The broker makes a database connection to the ODBC data source name (DSN) for each DSN even if different DSNs resolve to the same physical database.

The number of connections to a user database that a broker requires depends on the actions of the message flows that access the database. For each broker that accesses a database, one connection is required for every ODBC data source name (DSN) for each message flow thread. If a different node on the same thread uses the same DSN, the same connection is used, unless a different transaction mode is used, in which case another connection is required. This is explained further in "Database connections for coordinated message flows" on page 734.

When you start a broker, and while it is running, it opens connections to WebSphere MQ queues and to databases. The broker makes the connections when it needs to use them, and they remain open until:

- The message flow becomes idle

- The message flow is stopped

- The broker is stopped

Database connections from message flows that are not coordinated are released when a flow has no work. For example, a connection is released if there are no messages on a message flow's input queue, and the database has not been accessed for one minute.

On Windows, UNIX and Linux, to avoid breaking coordination, database connections are not released for coordinated message flows.

On z/OS database connections for coordinated message flows are released if the database has not been accessed for one minute.

If you are using the same database for user application data and for broker internal data, add the two connection requirements together when you calculate how many connections are required. For details of broker database connection requirements, see *Configuring access to databases*.

If you stop the broker, it releases all current database connections.

If you are using DB2 for your database, DB2's default action is to limit the number of concurrent connections to a database to the value of the *maxappls* configuration parameter. The default for *maxappls* is 40. If you believe that the connections that the broker might require exceed the value for *maxappls*, increase this and the associated parameter *maxagents* to new values based on your calculations.

If you are using another database, check the database documentation for information about connections and limits or restrictions.

When a message flow is idle, the execution group periodically releases database connections. Therefore, connections held by the broker reflect the broker's current use of these resources. This allows the broker to respond to database quiesce, where the database supports quiescing. Not all databases support the quiesce function, and not all databases quiesce in the same way. Check your database documentation for information about database quiescing. Also, see "Quiescing a database" and "Listing database connections that the broker holds" for more information.

Listing database connections that the broker holds

The broker does not have any functionality to list the connections it has to a database, instead use the facilities that your database supplies to list connections. Refer to the documentation for your database to find out about these.

Quiescing a database

This topic illustrates the behavior that WebSphere Message Broker expects when a database is quiesced. A database administrator issues the quiesce instruction on a database; it is not a function of the broker.

This topic assumes three things about the database being quiesced:

- The database can be quiesced
- New connections to the database are blocked by the database when it is quiescing
- Message flows using the database eventually become idle

The following list shows the behavior expected while a database is quiescing:

1. Tell the database to quiesce. As soon as you tell database to quiesce, connections that are in use remain in use, but no new connections to the database are allowed.
2. Processing messages. Messages that are using existing connections to the database continue to use their connection until the connection becomes idle. This can take a long time if messages continue to be processed. To ensure that messages are no longer processed, stop the message flow. Stopping the message flow stops messages being processed and releases the database connections that the flow was using. This ensures that the database connections that the flow holds become idle.

3. Database connections for the message flow become idle. This causes the broker to release the connections to the user databases that the message flow is using. When all connections to the database from the broker and from any other applications using the database are released, the database can complete its quiesce function.

User database DBCS restrictions and UNICODE support

The broker does not support DBCS-only columns within tables defined in user databases, therefore the following data types are not supported:

- DB2: GRAPHIC, VARGRAPHIC, LONGVARGRAPHIC
- Oracle: NCHAR, NVARCHAR, NCLOB
- Sybase: NCHAR, NVARCHAR

It is possible to manipulate UNICODE data in suitably configured databases but there are restrictions associated with this function; please refer to the readme.html file available on the product readmes Web page.

Data integrity within message flows

Code pages in which data is manipulated must be compatible between brokers and databases.

Subscription data retrieved from client applications (for example, topics from publishers and subscribers, and content filters from subscribers) and the character data entered through the workbench (for example, message flow names) are stored in the configuration repository. This data is translated from its originating code page to the code page of the process in which the broker or Configuration Manager is running, and then by the database manager to the code page in which the database or databases were created.

To preserve data consistency and integrity, ensure that all this subscription data and workbench character data is originated in a compatible code page to the two code pages to which it is translated. If you do not do so, you might get unpredictable results and lose data.

Data stored in the broker database is not affected in this way.

The restrictions described above do not apply to user data in messages. Ensure that any data in messages generated by your applications is compatible with the code page of any database you access from your message flows.

SQL statements generated as a result of explicit reference to databases within message processing nodes can contain character data that has a variety of sources. For example, the data might have been entered through the workbench, derived from message content, or read from another database. All this data is translated from its originating code page to the code page in which the broker was created, and then by the database manager to the code page in which the database was created. Ensure that these three code pages are compatible to avoid data conversion problems.

Validation properties for messages in the MRM domain

You can control validation by setting properties presented by the Validation tab and General Message Options tab on the following nodes:

Node type	Nodes with validation options
Input node	MQInput, SCADAInput, HTTPInput, JMSInput, TimeoutNotification
Output node	MQOutput, MQReply, SCADAOutput, HTTPReply, JMSOutput
Other nodes	Compute, Mapping, JavaCompute, Validate, ResetContentDescriptor, MQGet, HTTPRequest,

For an overview of message validation in the broker please refer to “Validating messages” on page 79.

The following validation properties can be set:

Tab	Properties that affect validation
Validate	Validate, Failure Action, Include All Value Constraints, Fix
General Message Options	Parse Timing

Validation tab properties

Validate

Sets whether validation is required. All nodes provide the following options:

None The default value. No validation is performed

Content

Indicates that you want to perform content checks, such as Content validation and Composition

Content and Value

Indicates that you want to perform content checks, such as Content validation and Composition, and value checks, such as whether the value conforms to data type, length, range, and enumeration.

Some nodes also provide the following option:

Inherit

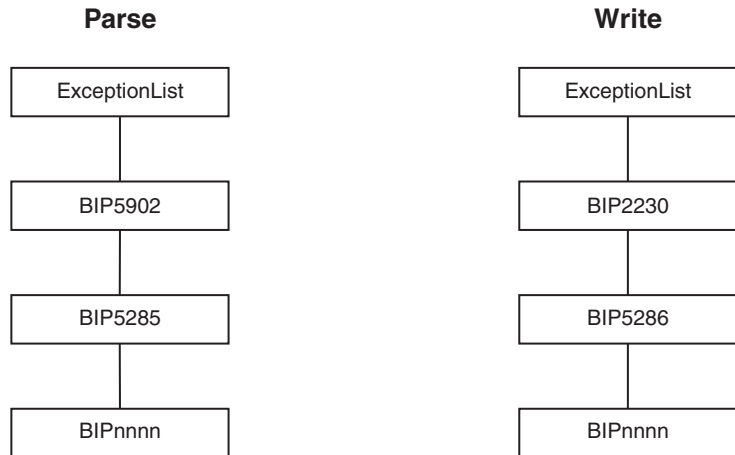
Instructs the node to use all the validation options provided with the input message tree in preference to any supplied on the node. Inherit will therefore resolve to one of None, Content, or Content and Value. If Inherit is selected, the other validation properties on the tab are greyed out.

Failure Action

The action that you want to be taken when a validation failure occurs. You can set it to the following values:

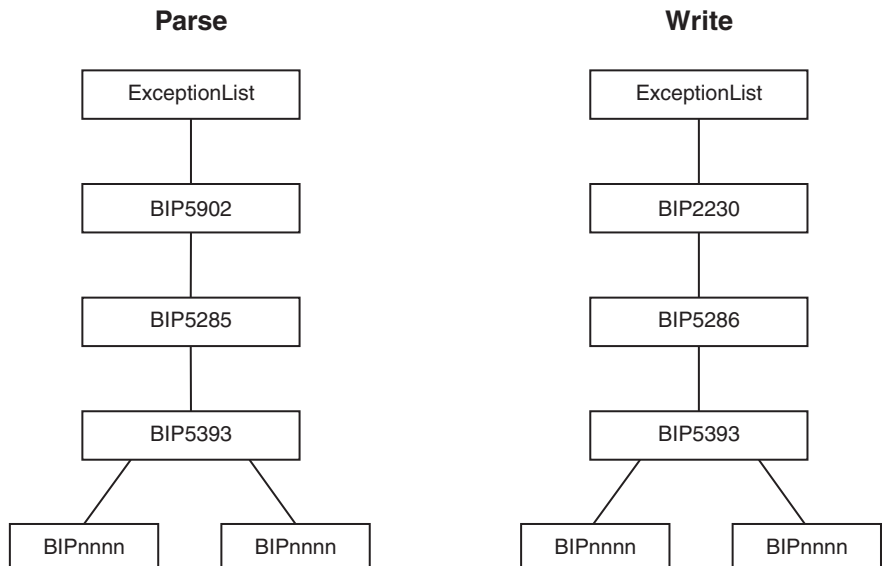
Exception

The default value. An exception is thrown on the first validation failure encountered. The resulting exception list is shown below. The failure is also logged in the user trace if you have asked for user tracing of the message flow, and validation stops. Use this setting if you want processing of the message to halt as soon as a failure is encountered.



Exception List

Throws an exception if validation failures are encountered, but only when the current parsing or writing operation has completed. The resulting exception list is shown below. Each failure is also logged in the user trace if you have asked for user tracing of the message flow, and validation stops. Use this setting if you want processing of the message to halt if a validation failure occurs, but you want to see the full list of failures encountered. Note that this property is affected by the Parse Timing property; when partial parsing is selected the current parsing operation will only parse a portion of an input message, so only the validation failures in that portion of the message will be reported.



User Trace

Logs all validation failures to the user trace, even if you have not asked for user tracing of the message flow. Use this setting if you want processing of the message to continue regardless of validation failures.

Local Error Log

Logs all validation failures to the error log (for example, the Event Log on Windows). Use this setting if you want processing of the message to continue regardless of validation failures.

Include All Value Constraints

The check box is selected. You cannot change this option.

Include All Value Constraints specifies which basic value constraint checks are to be performed on the values of fields in the message when *Validate* resolves to Content And Value. The checks performed are:

- Min Length
- Max Length
- Fraction Digits
- Total Digits
- Min Inclusive
- Max Inclusive
- Min Exclusive (where supported)
- Max Exclusive (where supported)
- Enumeration
- Pattern

For more details on value constraints, see Simple type logical value constraints.

Fix

The default value is none. You cannot change it.

None specifies that no extra remedial action is to be taken when *Validate* resolves to Content or Content And Value and validation failures occur. The remedial action taken depends on the setting of *Failure Action*.

If *Failure Action* is set to User Trace or Local Error Log, the remedial action is limited to the default remedial action that takes place when no validation is being performed, as described in “Validating messages” on page 79:

1. Extraneous fields are discarded on output for fixed formats (CWF and TDS fixed length models only)
2. If mandatory content is missing, defaults are supplied, if available, on output for fixed formats (CWF and TDS fixed length models only)
3. If an element’s data type in the tree does not match that specified in the dictionary the data type is converted on output to match the dictionary definition if possible, for all formats.

If *Failure Action* is set to Exception or Exception List, the remedial action is limited to the third item in the above list, and an exception is thrown for the first two items.

Parse Timing

The Parse Timing property is only available on the following nodes:

Node type	Nodes containing Parse Timing property options
Input nodes	MQInput, MQGet, SCADAInput, HTTPInput, HTTPRequest, Real-timeInput, JMSInput, Compute JavaCompute, Mapping, ResetContentDescriptor, TimeoutNotification
Other nodes	ResetContentDescriptor

The *Parse Timing* property determines whether on demand parsing is to be used when parsing a message. It also gives you control over the timing of MRM input message validation:

- If you select a Parse Timing value of On Demand, validation of a field in the message is delayed until it is parsed by partial parsing.
- If you select a Parse Timing value of Immediate, partial parsing is overridden, and everything in the message is parsed and validated except those complex types with a Composition of Choice or Message that can not be resolved at the time
- If you select a Parse Timing value of Complete, partial parsing is overridden, and everything is parsed and validated. Complex types with a Composition of Choice or Message that can not be resolved at the time cause a validation failure.

If you switch on MRM message validation, and you select On Demand or Immediate for Parse Timing, validation errors might not be detected until later in the processing of a message by a message flow, or might never be detected if a portion of the message is never parsed. To make sure that all fields in a message are validated, either select Complete, or select Immediate and make sure that you resolve all unresolved types with a Composition of Choice or Message at the start of your message flow.

The Parse Timing property has no effect on the validation of output messages.

Parsing on demand

If a parser is capable of parsing an input bit stream on demand, instead of immediately parsing the entire bit stream, the *Parse Timing* property of a message flow node controls the on demand behavior of the parser.

On demand parsing is referred to in the message broker as partial parsing. The parsers that are capable of performing partial parsing of input messages are the MRM, XML, XMLNS and XMLNSC parsers. Additionally, for the MRM parser, because input message validation is performed during parsing, the *Parse Timing* property also has an effect on validation.

You can set the *Parse Timing* property to On Demand (the default), Immediate, or Complete.

On Demand causes partial parsing to occur. When fields in the message are referenced, as much of the message is parsed as is necessary to completely resolve the reference. Fields may therefore not be parsed until late in the message flow, or never. This applies to both the message body and the message headers.

Immediate and Complete both override partial parsing and parse the entire message including any message headers, except when the MRM parser encounters an element with a complex type of Composition Choice or Message that can not be resolved at the time; for example, the content needs to be resolved by the user in ESQL. For a Choice, the data is added to the message tree as an unresolved item and parsing continues with the next element. For a Message, parsing terminates at that point. The only difference in behavior between Immediate and Complete occurs when MRM validation is enabled.

The *Parse Timing* property also gives you control over how MRM message validation interacts with partial parsing. Refer to “Validation properties for messages in the MRM domain” on page 703 for a full description.

The *Parse Timing* property has no effect on the serialization of output messages.

Exception list structure

The following figure shows one way in which to construct an exception list.

```

ExceptionList {
  RecoverableException = {
    1
    File = 'f:/build/argo/src/DataFlowEngine/ImbDataFlowNode.cpp'
    Line = 538
    Function = 'ImbDataFlowNode::createExceptionList'
    Type = 'ComIbmComputeNode'
    Name = '0e416632-de00-0000-0080-bdb4d59524d5'
    Label = 'mf1.Compute1'
    Text = 'Node throwing exception'
    Catalog = 'WebSphere Message Broker2'
    Severity = 3
    Number = 2230
    RecoverableException = {
      2
      File = 'f:/build/argo/src/DataFlowEngine/ImbRdlBinaryExpression.cpp'
      Line = 231
      Function = 'ImbRdlBinaryExpression::scalarEvaluate'
      Type = 'ComIbmComputeNode'
      Name = '0e416632-de00-0000-0080-bdb4d59524d5'
      Label = 'mf1.Compute1'
      Text = 'error evaluating expression'
      Catalog = 'WebSphere Message Broker2'
      Severity = 2
      Number = 2439
      Insert = {
        Type = 2
        Text = '2'
      }
      Insert = {
        Type = 2
        Text = '30'
      }
    }
    RecoverableException = {
      3
      File = 'f:/build/argo/src/DataFlowEngine/ImbRdlValueOperations.cpp'
      Line = 257
      Function = 'intDivideInt'
      Type = 'ComIbmComputeNode'
      Name = '0e416632-de00-0000-0080-bdb4d59524d5'
      Label = 'mf1.Compute1'
      Text = 'Divide by zero calculating '%1 / %2''
      Catalog = 'WebSphere Message Broker2'
      Severity = 2
      Number = 2450
      Insert = {
        Type = 5
        Text = '100 / 0'
      }
    }
  }
}
}
}

```

Notes:

1. The first exception description 1 is a child of the root. This identifies error number 2230, indicating that an exception has been thrown. The node that has thrown the exception is also identified (mf1.Compute1).
2. Exception description 2 is a child of the first exception description 1. This identifies error number 2439.
3. Exception description 3 is a child of the second exception description 2. This identifies error number 2450, which indicates that the node has attempted to divide by zero.

The following topics provide examples of exception lists that have been written to the trace output destination (by the Trace node):

- “Database exception trace output”
- “Conversion exception trace output” on page 711
- “Parser exception trace output” on page 713
- “User exception trace output” on page 713

Database exception trace output

The following figure shows an extract of the output that might be generated by a Trace node that has its property *Pattern* set to a value that represents a structure that includes the ExceptionList tree.

The exception shown occurred when a database exception was detected

```
ExceptionList = (
  (0x1000000)RecoverableException = (
    (0x3000000)File           = 'F:\build\S000_D\src\DataFlowEngine\ImbComputeNode.cpp'
    (0x3000000)Line           = 402
    (0x3000000)Function        = 'ImbComputeNode::evaluate'
    (0x3000000)Type            = 'ComIbmComputeNode'
    (0x3000000)Name            = 'acd8f35d-e700-0000-0080-b78796c5e70d'
    (0x3000000)Label          = 'esql_13485_check_defect.Compute1'
    (0x3000000)Text            = 'Caught exception and rethrowing'
    (0x3000000)Catalog        = 'WMQIv210'
    (0x3000000)Severity        = 3
    (0x3000000)Number          = 2230
  )
  (0x1000000)RecoverableException = (
    (0x3000000)File           = 'F:\build\S000_D\src\DataFlowEngine\ImbRdl\ImbRdlExternalDb.cpp'
    (0x3000000)Line           = 278
    (0x3000000)Function        = 'SqlExternalDbStmt::executeStmt'
    (0x3000000)Type            = 'ComIbmComputeNode'
    (0x3000000)Name            = 'acd8f35d-e700-0000-0080-b78796c5e70d'
    (0x3000000)Label          = 'esql_13485_check_defect.Compute1'
    (0x3000000)Text            = 'The following error occurred execution SQL statement &3. inserts where &4'
    (0x3000000)Catalog        = 'WMQIv210'
    (0x3000000)Severity        = 3
    (0x3000000)Number          = 2519
  )
  (0x1000000)Insert           = (
    (0x3000000)Type = 2
    (0x3000000)Text = '1'
  )
  (0x1000000)Insert           = (
    (0x3000000)Type = 2
    (0x3000000)Text = '1'
  )
  (0x1000000)Insert           = (
    (0x3000000)Type = 5
    (0x3000000)Text = 'USERDB'
  )
  (0x1000000)Insert           = (
    (0x3000000)Type = 5
    (0x3000000)Text = 'DELETE FROM DB2ADMIN.STOCK WHERE (STOCK_ID)=(?)'
  )
  (0x1000000)Insert           = (
    (0x3000000)Type = 5
    (0x3000000)Text = '500027, '
  )
  (0x1000000)DatabaseException = (
    (0x3000000)File           = 'F:\build\S000_D\src\DataFlowEngine\ImbOdbc.cpp'
    (0x3000000)Line           = 153
    (0x3000000)Function        = 'ImbOdbcHandle::checkRcInner'
    (0x3000000)Type            = ''
    (0x3000000)Name            = ''
    (0x3000000)Label          = ''
    (0x3000000)Text            = 'Root SQL exception'
    (0x3000000)Catalog        = 'WMQIv210'
    (0x3000000)Severity        = 3
    (0x3000000)Number          = 2321
  )
  (0x1000000)Insert           = (
    (0x3000000)Type = 2
    (0x3000000)Text = '100'
  )
)
)
)
)
```

Conversion exception trace output

The following figure shows an extract of the output that might be generated by a Trace node that has its property *Pattern* set to a value that represents a structure that includes the ExceptionList tree.

The exception shown occurred when a conversion (CAST) exception was detected.

```
ExceptionList = (  
  (0x1000000) RecoverableException = (  
    (0x3000000) File = 'F:\build\S000_D\src\DataFlowEngine\ImbComputeNode.cpp'  
    (0x3000000) Line = 402  
    (0x3000000) Function = 'ImbComputeNode::evaluate'  
    (0x3000000) Type = 'ComIbmComputeNode'  
    (0x3000000) Name = 'acd8f35d-e700-0000-0080-b78796c5e70d'  
    (0x3000000) Label = 'esql_13485_check_defect.Compute1'  
    (0x3000000) Text = 'Caught exception and rethrowing'  
    (0x3000000) Catalog = 'WMQIv210'  
    (0x3000000) Severity = 3  
    (0x3000000) Number = 2230  
    (0x1000000) RecoverableException = (  
      (0x3000000) File = 'F:\build\S000_D\src\DataFlowEngine\ImbRdl\ImbRdlTypeCast.cpp'  
      (0x3000000) Line = 163  
      (0x3000000) Function = 'SqlTypeCast::evaluate'  
      (0x3000000) Type = ''  
      (0x3000000) Name = ''  
      (0x3000000) Label = ''  
      (0x3000000) Text = 'Error casting from %3 to %4'  
      (0x3000000) Catalog = 'WMQIv210'  
      (0x3000000) Severity = 3  
      (0x3000000) Number = 2521  
      (0x1000000) Insert = (  
        (0x3000000) Type = 2  
        (0x3000000) Text = '12'  
      )  
    (0x1000000) Insert = (  
      (0x3000000) Type = 2  
      (0x3000000) Text = '28'  
    )  
    (0x1000000) Insert = (  
      (0x3000000) Type = 5  
      (0x3000000) Text = 'CHARACTER'  
    )  
    (0x1000000) Insert = (  
      (0x3000000) Type = 5  
      (0x3000000) Text = 'INTEGER'  
    )  
    (0x1000000) ConversionException = (  
      (0x3000000) File = 'F:\build\S000_D\src\CommonServices\ImbUtility.cpp'  
      (0x3000000) Line = 195  
      (0x3000000) Function = 'imbWcsToInt64'  
      (0x3000000) Type = ''  
      (0x3000000) Name = ''  
      (0x3000000) Label = ''  
      (0x3000000) Text = 'Invalid characters'  
      (0x3000000) Catalog = 'WMQIv210'  
      (0x3000000) Severity = 3  
      (0x3000000) Number = 2595  
      (0x1000000) Insert = (  
        (0x3000000) Type = 5  
        (0x3000000) Text = 'fred'  
      )  
    )  
  )  
)  
)  
)  
)  
)  
)
```


Parser exception trace output

The following figure shows an extract of the output that might be generated by a Trace node that has its property *Pattern* set to a value that represents a structure that includes the ExceptionList tree.

The exception shown occurred when the message parser was invoked.

```
ExceptionList = (  
  (0x1000000) RecoverableException = (  
    (0x3000000) File = 'F:\build\S000_D\src\DataFlowEngine\ImbMqOutputNode.cpp'  
    (0x3000000) Line = 1444  
    (0x3000000) Function = 'ImbMqOutputNode::evaluate'  
    (0x3000000) Type = 'ComIbmMQOutputNode'  
    (0x3000000) Name = 'c76eb6cd-e600-0000-0080-b78796c5e70d'  
    (0x3000000) Label = 'esql_13485_check_defect.OUT'  
    (0x3000000) Text = 'Caught exception and rethrowing'  
    (0x3000000) Catalog = 'WMQIv210'  
    (0x3000000) Severity = 3  
    (0x3000000) Number = 2230  
    (0x1000000) ParserException = (  
      (0x3000000) File = 'F:\build\S000_D\src\MTI\MTIforBroker\GenXmlParser2\XmlImbParser.cpp'  
      (0x3000000) Line = 210  
      (0x3000000) Function = 'XmlImbParser::refreshBitStreamFromElements'  
      (0x3000000) Type = 'ComIbmMQInputNode'  
      (0x3000000) Name = 'ce64b6cd-e600-0000-0080-b78796c5e70d'  
      (0x3000000) Label = 'esql_13485_check_defect.IN'  
      (0x3000000) Text = 'XML Writing Errors have occurred'  
      (0x3000000) Catalog = 'WMQIv210'  
      (0x3000000) Severity = 3  
      (0x3000000) Number = 5010  
      (0x1000000) ParserException = (  
        (0x3000000) File = 'F:\build\S000_D\src\MTI\MTIforBroker\GenXmlParser2\XmlImbParser.cpp'  
        (0x3000000) Line = 551  
        (0x3000000) Function = 'XmlImbParser::checkForBodyElement'  
        (0x3000000) Type = ''  
        (0x3000000) Name = ''  
        (0x3000000) Label = ''  
        (0x3000000) Text = 'No valid body of the document could be found.'  
        (0x3000000) Catalog = 'WMQIv210'  
        (0x3000000) Severity = 3  
        (0x3000000) Number = 5005  
      )  
    )  
  )  
)
```

User exception trace output

The following figure shows an extract of the output that might be generated by a Trace node that has its property *Pattern* set to a value that represents a structure that includes the ExceptionList tree.

The exception shown occurred when a user exception was generated (with the ESQL THROW statement).

```

ExceptionList = (
  (0x1000000)RecoverableException = (
    (0x3000000)File           = 'F:\build\S000_D\src\DataFlowEngine\ImbComputeNode.cpp'
    (0x3000000)Line           = 402
    (0x3000000)Function        = 'ImbComputeNode::evaluate'
    (0x3000000)Type            = 'ComIbmComputeNode'
    (0x3000000)Name            = 'acd8f35d-e700-0000-0080-b78796c5e70d'
    (0x3000000)Label           = 'esql_13485_check_defect.Compute1'
    (0x3000000)Text            = 'Caught exception and rethrowing'
    (0x3000000)Catalog         = 'WMQIv210'
    (0x3000000)Severity        = 3
    (0x3000000)Number          = 2230
  (0x1000000)UserException = (
    (0x3000000)File           = 'F:\build\S000_D\src\DataFlowEngine\ImbRdl\ImbRdlThrowExceptionStatements.cpp'
    (0x3000000)Line           = 148
    (0x3000000)Function        = 'SqlThrowExceptionStatement::execute'
    (0x3000000)Type            = 'ComIbmComputeNode'
    (0x3000000)Name            = 'acd8f35d-e700-0000-0080-b78796c5e70d'
    (0x3000000)Label           = 'esql_13485_check_defect.Compute1'
    (0x3000000)Text            = 'User Generated SQL 'USER' exception'
    (0x3000000)Catalog         = 'WMQIv210'
    (0x3000000)Severity        = 1
    (0x3000000)Number          = 2949
    (0x1000000)Insert          = (
      (0x3000000)Type = 5
      (0x3000000)Text = 'USER'
    )
    (0x1000000)Insert          = (
      (0x3000000)Type = 5
      (0x3000000)Text = 'Insert1'
    )
    (0x1000000)Insert          = (
      (0x3000000)Type = 5
      (0x3000000)Text = 'Insert2'
    )
    (0x1000000)Insert          = (
      (0x3000000)Type = 5
      (0x3000000)Text = 'etc'
    )
    (0x1000000)Insert          = (
      (0x3000000)Type = 5
      (0x3000000)Text = ''
    )
    (0x1000000)Insert          = (
      (0x3000000)Type = 5
      (0x3000000)Text = ''
    )
    (0x1000000)Insert          = (
      (0x3000000)Type = 5
      (0x3000000)Text = ''
    )
    (0x1000000)Insert          = (
      (0x3000000)Type = 5
      (0x3000000)Text = ''
    )
  )
)
)
)

```

Configurable message flow properties

When you add a message flow to a broker archive (bar) file in preparation for deploying it to a broker, you can set additional properties that influence its runtime operation. These properties are available for review and update when you select the **Configure** tab for the broker archive file.

Additional Instances

Specifies the number of additional threads that the broker can use to service the message flow. These additional threads are created only if there are sufficient input messages. You can have up to 256 threads. The default value is 0. Additional threads can increase the throughput of a message flow but you should consider the potential impact on message order.

If the message flow processes WebSphere MQ messages, you can configure the message flow to control the message order. Set the *Order Mode* property on the MQInput node accordingly. You might also need to set the *Commit by Message Group* and *Logical Order* properties.

The broker opens the input queue as shared (using the MQOO_INPUT_SHARED option), so you must ensure that the input queue has been defined with the SHARE property to enable multiple broker threads to read from the same input queue.

For more information about the node properties, refer to the “MQInput node” on page 593.

Commit Count

Specifies how many input WebSphere MQ messages are processed by a message flow before a syncpoint is taken (by issuing an MQCMIT).

Set this property only if you have set *Additional Instances* to 0.

The default value of 1 is also the minimum permitted value. Change this property to avoid frequent MQCMIT calls when messages are being processed quickly and the lack of an immediate commit can be tolerated by the receiving application.

Use the *Commit Interval* to ensure that a commit is performed periodically when not enough messages are received to fulfill the *Commit Count*.

This property has no effect if the message flow does not process WebSphere MQ messages.

Commit Interval

For WebSphere MQ messages, specifies a time interval at which a commit is taken when the *Commit Count* property is greater than 1 (that is, where the message flow is batching messages), but the number of messages processed has not reached the value of the *Commit Count* property. It ensures that a commit is performed periodically when not enough messages are received to fulfill the *Commit Count*.

The time interval is specified in seconds, as a decimal number with a maximum of 3 decimal places (millisecond granularity). The value must be in the range 0.000 through 60.000. The default value is 0.

Set this property only if you have set *Additional Instances* to 0.

This property has no effect if the message flow does not process WebSphere MQ messages.

Coordinated Transaction

Controls whether the message flow is processed as a global transaction, coordinated by WebSphere MQ. Such a message flow is said to be fully globally-coordinated. The default value is No.

Use coordinated transactions only where you need to process the message and any database updates performed by the message flow in a single unit-of-work, using a two-phase commit protocol. This means that both the message is read and the database updates are performed, or neither is done.

If you change this value, ensure that the broker's queue manager is configured correctly. If you do not set up the queue manager correctly, the broker generates a message when the message flow receives a message to indicate that, although the message flow is to be globally coordinated, the queue manager configuration does not support this.

See Supported databases for information about which databases are supported as participants in a global transaction, and the *WebSphere MQ System Administration* book for how to configure WebSphere MQ and the database managers.

This property has no effect if the message flow does not process WebSphere MQ messages.

User-defined properties

A user-defined property (UDP) is a user-defined constant whose initial value can be modified, at design time, by the Message Flow editor, or overridden, at deployment time, by the Broker Archive editor. The advantage of UDPs is that their values can be changed by operational staff at deployment time. If, for example, you use UDPs to hold configuration data, you can configure a message flow for a particular machine, task, or environment at deployment time, without having to change the code at the node level.

For introductory information about UDPs, see "User-defined properties in ESQL" on page 149.

For information about configuring UDPs at deployment time, see "Configuring a message flow at deployment time using UDPs" on page 284.

You can view and update other configurable properties for the message flow. The properties that are displayed depend on the nodes within the message flow; some have no configurable properties to display. The node properties that are configurable are predominantly system-related properties that are likely to change for each broker to which the message flow is deployed. These properties include data source names and the names of WebSphere MQ queues and queue managers. For full details of configurable properties for a node, see the appropriate node description.

Message flow porting considerations

If you have configured a message flow that runs on a broker on a distributed system, and you now want to deploy it to a broker that runs on z/OS, be aware of the following:

WebSphere MQ queue manager and queue names

There are restrictions on WebSphere MQ resource names on z/OS:

- The queue manager name cannot be greater than four characters.
- All queue names must be in uppercase. Although using quotation marks preserves the case, certain WebSphere MQ activities on z/OS cannot find the queue names being referenced.

For more information about configuring on z/OS, refer to the *WebSphere MQ for z/OS Concepts and Planning Guide*.

File system references

File system references must reflect a UNIX file path. If you deploy a message flow to z/OS that you have previously run on Windows, you might have to make changes. If you have previously deployed the message flow to a UNIX system (AIX, Linux, Solaris, or HP-UX), you do not have to make any changes.

Databases

If the message flow accesses one or more databases, it might be subject to some restrictions based on the system on which the database is defined. These restrictions are described in Database locations.

Message flow accounting and statistics data

This section provides information for message flow accounting and statistics data.

Details of the information that is collected, and the output formats in which it can be recorded, are provided:

- Statistics details
- Data formats
- Example output

You can also find information on how to use accounting and statistics data to improve the performance of a message flow in this developerWorks article on message flow performance.

Message flow accounting and statistics details

This topic identifies the statistics that are collected for message flows.

The details that are available are:

Message flow statistics

One record is created for each message flow in an execution group. Each record contains the following details:

- Message flow name and UUID
- Execution group name and UUID
- Broker name and UUID
- Start and end times for data collection
- Type of data collected (snapshot or archive)
- CPU and elapsed time spent processing messages
- CPU and elapsed time spent waiting for input
- Number of messages processed
- Minimum, maximum, and average message sizes
- Number of threads available and maximum assigned at any time
- Number of messages committed and backed out
- Accounting origin

Thread statistics

One record is created for each thread assigned to the message flow. Each record contains the following details:

- Thread number (this has no significance and is for identification only)
- CPU and elapsed time spent processing messages
- CPU and elapsed time spent waiting for input
- Number of messages processed
- Minimum, maximum, and average message sizes

Node statistics

One record is created for each node in the message flow. Each record contains the following details:

- Node name
- Node type (for example MQInput)
- CPU time spent processing messages
- Elapsed time spent processing messages
- Number of times that the node is invoked
- Number of messages processed
- Minimum, maximum, and average message sizes

Terminal statistics

One record is created for each terminal on a node. Each record contains the following details:

- Terminal name
- Terminal type (input or output)
- Number of times that a message is propagated through this terminal

For further details about specific output formats, see the following topics:

- “User trace entries for message flow accounting and statistics data” on page 722
- “XML publication for message flow accounting and statistics data”
- “z/OS SMF records for message flow accounting and statistics data” on page 726

Message flow accounting and statistics output formats

The message flow accounting and statistics data is written in one of three formats:

- User trace entries
- XML publication
- z/OS SMF records

XML publication for message flow accounting and statistics data

This topic describe the information that is written to the XML publication for message flow accounting and statistics data. The data is created within the folder `WMQIStatisticsAccounting`, which contains subfolders that provide more detailed information. All folders are present within the publication even if you set current data collection parameters to specify that the relevant data is not collected.

Snapshot data is used for performance analysis, and is published as retained and non-persistent. Archive data is used for accounting where an audit trail might be required, and is published as retained and persistent. All publications are global and can be collected by a subscriber that has registered anywhere in the network. They can also be collected by more than one subscriber.

One XML publication is generated for each message flow that is producing data for the time period you have chosen. For example, if `MessageFlowA` and

MessageFlowB, are both producing archive data over a period of 60 minutes, both MessageFlowA and MessageFlowB will produce an XML publication every 60 minutes.

If you are concerned about the safe delivery of these messages, for example for charging purposes, use a secure delivery mechanism such as WebSphere MQ.

The folders and subfolders in the XML publication have the following identifiers:

- WMQIStatisticsAccounting
- MessageFlow
- Threads
- ThreadStatistics
- Nodes
- NodesStatistics
- TerminalStatistics

The tables provided here describe the contents of each of these folders in the order listed above.

The table below describes the general accounting and statistics information, created in folder WMQIStatisticsAccounting.

Field	Data type	Details
RecordType	Character	Type of output, one of: <ul style="list-style-type: none"> • Archive • Snapshot
RecordCode	Character	Reason for output, one of: <ul style="list-style-type: none"> • MajorInterval • Snapshot • Shutdown • ReDeploy • StatsSettingsModified

The table below describes the message flow statistics information, created in folder MessageFlow.

Field	Data type	Details
BrokerLabel	Character (maximum 32)	Broker name
BrokerUUID	Character (maximum 32)	Broker universal unique identifier
ExecutionGroupName	Character (maximum 32)	Execution group name
ExecutionGroupUUID	Character (maximum 32)	Execution group universal unique identifier
MessageFlowName	Character (maximum 32)	Message flow name
StartDate	Character	Interval start date (YYYY-MM-DD)
StartTime	Character	Interval start time (HH:MM:SS:NNNNNN)
EndDate	Character	Interval end date (YYYY-MM-DD)

Field	Data type	Details
EndTime	Character	Interval end time (HH:MM:SS:NNNNNN)
TotalElapsedTime	Numeric	Total elapsed time spent processing input messages (microseconds)
MaximumElapsedTime	Numeric	Maximum elapsed time spent processing an input message (microseconds)
MinimumElapsedTime	Numeric	Minimum elapsed time spent processing an input message (microseconds)
TotalCPUTime	Numeric	Total CPU time spent processing input messages (microseconds)
MaximumCPUTime	Numeric	Maximum CPU time spent processing an input message (microseconds)
MinimumCPUTime	Numeric	Minimum CPU time spent processing an input message (microseconds)
CPUTimeWaitingForInputMessage	Numeric	Total CPU time spent waiting for input messages (microseconds)
ElapsedTimeWaitingForInputMessage	Numeric	Total elapsed time spent waiting for input messages (microseconds)
TotalInputMessages	Numeric	Total number of messages processed
TotalSizeOfInputMessages	Numeric	Total size of input messages (bytes)
MaximumSizeOfInputMessages	Numeric	Maximum input message size (bytes)
MinimumSizeOfInputMessages	Numeric	Minimum message input size (bytes)
NumberOfThreadsInPool	Numeric	Number of threads in pool
TimesMaximumNumberOfThreadsReached	Numeric	Number of times the maximum number of threads is reached
TotalNumberOfMQErrors ¹	Numeric	Number of MQGET errors (MQInput node) or Web services errors (HTTPInput node)
TotalNumberOfMessagesWithErrors ²	Numeric	Number of messages that contain errors
TotalNumberOfErrorsProcessingMessages	Numeric	Number of errors processing a message
TotalNumberOfTimeOutsWaitingForRepliesToAggregateMessages	Numeric	Number of timeouts processing a message (AggregateReply node only)

Field	Data type	Details
TotalNumberOfCommits	Numeric	Number of transaction commits
TotalNumberOfBackouts	Numeric	Number of transaction backouts
AccountingOrigin	Character (maximum 32)	Accounting origin
Notes: 1. For example, a conversion error occurs when the message is got from the queue. 2. These include exceptions that are thrown downstream of the input node, and errors detected by the input node after it has successfully retrieved the message from the queue but before it has propagated it to the out terminal (for example, a format error).		

The table below describes the thread statistics information, created in folder Threads.

Field	Data type	Details
Number	Numeric	Number of thread statistics subfolders within Threads folder

The table below describes the thread statistics information for each individual thread, created in folder ThreadStatistics, a subfolder of Threads.

Field	Data type	Details
Number	Numeric	Relative thread number in pool
TotalNumberOfInputMessages	Numeric	Total number of messages processed by thread
TotalElapsedTime	Numeric	Total elapsed time spent processing input messages (microseconds)
TotalCUPTime	Numeric	Total CPU time spent processing input messages (microseconds)
CPUTimeWaitingForInputMessage	Numeric	Total CPU time spent waiting for input messages (microseconds)
ElapsedTimeWaitingForInputMessage	Numeric	Total elapsed time spent waiting for input messages (microseconds)
TotalSizeOfInputMessages	Numeric	Total size of input messages (bytes)
MaximumSizeOfInputMessages	Numeric	Maximum size of input messages (bytes)
MinimumSizeOfInputMessages	Numeric	Minimum size of input messages (bytes)

The table below describes the node statistics information, created in folder Nodes.

Field	Data type	Details
Number	Numeric	Number of node statistics subfolders within Nodes folder

The table below describes the node statistics information for each individual node, created in folder NodesStatistics, a subfolder of Nodes.

Field	Data type	Details
Label	Character	Name of node (Label)
Type	Character	Type of node
TotalElapsedTime	Numeric	Total elapsed time spent processing input messages (microseconds)
MaximumElapsedTime	Numeric	Maximum elapsed time spent processing input messages (microseconds)
MinimumElapsedTime	Numeric	Minimum elapsed time spent processing input messages (microseconds)
TotalCPUTime	Numeric	Total CPU time spent processing input messages (microseconds)
MaximumCPUTime	Numeric	Maximum CPU time spent processing input messages (microseconds)
MinimumCPUTime	Numeric	Minimum CPU time spent processing input messages (microseconds)
CountOfInvocations	Numeric	Total number of messages processed by this node
NumberOfInputTerminals	Numeric	Number of input terminals
NumberOfOutputTerminals	Numeric	Number of output terminals

The table below describes the terminal statistics information, created in folder TerminalStatistics.

Field	Data type	Details
Label	Character	Name of terminal
Type	Character	Type of terminal, one of: <ul style="list-style-type: none"> • Input • Output
CountOfInvocations	Numeric	Total number of invocations

User trace entries for message flow accounting and statistics data

This topic describes the information that is written to the user trace log for message flow accounting and statistics data.

The data records are identified by the following message numbers:

- BIP2380I
- BIP2381I
- BIP2382I
- BIP2383I

The inserts for each message are described in the following tables, in the order shown above.

The following table describes the inserts in message BIP2380I. One message is written for the message flow.

Field	Data type	Details
ProcessID	Numeric	Process ID

Field	Data type	Details
Key	Numeric	Key used to associate related accounting and statistics BIP messages
Type	Character	Type of output, one of: <ul style="list-style-type: none"> • Archive • Snapshot
Reason	Character	Reason for output, one of: <ul style="list-style-type: none"> • MajorInterval • Snapshot • Shutdown • ReDeploy • StatsSettingsModified
BrokerLabel	Character (maximum 32)	Broker name
BrokerUUID	Character (maximum 32)	Broker universal unique identifier
ExecutionGroupName	Character (maximum 32)	Execution group name
ExecutionGroupUUID	Character (maximum 32)	Execution group universal unique identifier
MessageFlowName	Character (maximum 32)	Message flow name
StartDate	Character	Interval start date (YYYY-MM-DD)
StartTime	Character	Interval start time (HH:MM:SS:NNNNNN)
EndDate	Character	Interval end date (YYYY-MM-DD)
EndTime	Character	Interval end time (HH:MM:SS:NNNNNN)
TotalElapsedTime	Numeric	Total elapsed time spent processing input messages (microseconds)
MaximumElapsedTime	Numeric	Maximum elapsed time spent processing an input message (microseconds)
MinimumElapsedTime	Numeric	Minimum elapsed time spent processing an input message (microseconds)
TotalCPUTime	Numeric	Total CPU time spent processing input messages (microseconds)
MaximumCPUTime	Numeric	Maximum CPU time spent processing an input message (microseconds)
MinimumCPUTime	Numeric	Minimum CPU time spent processing an input message (microseconds)
CPUTimeWaitingForInputMessage	Numeric	Total CPU time spent waiting for input messages (microseconds)

Field	Data type	Details
ElapsedTimeWaitingForInputMessage	Numeric	Total elapsed time spent waiting for input messages (microseconds)
TotalInputMessages	Numeric	Total number of messages processed
TotalSizeOfInputMessages	Numeric	Total size of input messages (bytes)
MaximumSizeOfInputMessages	Numeric	Maximum input message size (bytes)
MinimumSizeOfInputMessages	Numeric	Minimum input message size (bytes)
NumberOfThreadsInPool	Numeric	Number of threads in pool
TimesMaximumNumberofThreadsReached	Numeric	Number of times the maximum number of threads is reached
TotalNumberOfMQErrors ¹	Numeric	Number of MQGET errors (MQInput node) or Web services errors (HTTPInput node)
TotalNumberOfMessagesWithErrors ²	Numeric	Number of messages that contain errors
TotalNumberOfErrorsProcessingMessages	Numeric	Number of errors processing a message
TotalNumberOfTimeOutsWaitingForRepliesToAggregateMessages	Numeric	Number of timeouts processing a message (AggregateReply node only)
TotalNumberOfCommits	Numeric	Number of transaction commits
TotalNumberOfBackouts	Numeric	Number of transaction backouts
AccountingOrigin	Character (maximum 32)	Accounting origin
Notes:		
1. For example, a conversion error occurs when the message is got from the queue.		
2. These include exceptions that are thrown downstream of the input node, and errors detected by the input node after it has successfully retrieved the message from the queue (for example, a format error).		

The following table describes the inserts in message BIP2381I. One message is written for each thread.

Field	Data type	Details
ProcessID	Numeric	Process ID
Key	Numeric	Key used to associate related accounting and statistics BIP messages
Number	Numeric	Relative thread number in pool
TotalNumberOfInputMessages	Numeric	Total number of messages processed by thread
TotalElapsedTime	Numeric	Total elapsed time spent processing input messages (microseconds)

Field	Data type	Details
TotalCUPTime	Numeric	Total CPU time spent processing input messages (microseconds)
CPUTimeWaitingForInputMessage	Numeric	Total CPU time spent waiting for input messages (microseconds)
ElapsedTimeWaitingForInputMessage	Numeric	Total elapsed time spent waiting for input messages (microseconds)
TotalSizeOfInputMessages	Numeric	Total size of input messages (bytes)
MaximumSizeOfInputMessages	Numeric	Maximum size of input messages (bytes)
MinimumSizeOfInputMessages	Numeric	Minimum size of input messages (bytes)

The following table describes the inserts in message BIP2382I. One message is written for each node.

Field	Data type	Details
ProcessID	Numeric	Process ID
Key	Numeric	Key used to associate related accounting and statistics BIP messages
Label	Character	Name of node (Label)
Type	Character	Type of node
TotalElapsedTime	Numeric	Total elapsed time spent processing input messages (microseconds)
MaximumElapsedTime	Numeric	Maximum elapsed time spent processing input messages (microseconds)
MinimumElapsedTime	Numeric	Minimum elapsed time spent processing input messages (microseconds)
TotalCPUTime	Numeric	Total CPU time spent processing input messages (microseconds)
MaximumCPUTime	Numeric	Maximum CPU time spent processing input messages (microseconds)
MinimumCPUTime	Numeric	Minimum CPU time spent processing input messages (microseconds)
CountOfInvocations	Numeric	Total number of messages processed by this node
NumberOfInputTerminals	Numeric	Number of input terminals
NumberOfOutputTerminals	Numeric	Number of output terminals

The following table describes the inserts in message BIP2383I. One message is written for each terminal on each node.

Field	Data type	Details
ProcessID	Numeric	Process ID
Key	Numeric	Key used to associate related accounting and statistics BIP messages
Label	Character	Name of terminal
Type	Character	Type of terminal, one of: <ul style="list-style-type: none"> • Input • Output

Field	Data type	Details
CountOfInvocations	Numeric	Total number of invocations

z/OS SMF records for message flow accounting and statistics data

This topic describes the information that is written to z/OS SMF records for message flow accounting and statistics data.

The data records are type 117 records with the following identifiers:

- BipSMFDate
- BipSMFRecordHdr
- BipSMFTriplet
- BipSMFMessageFlow
- BipSMFThread
- BipSMFNode
- BipSMFTerminal

The following tables describe the contents of each of these records in the order listed above.

The following table describes the contents of the BipSMFDate record.

Field	Data type	Details
YYYY	signed short int	2 byte year
MM	char	1 byte month
DD	char	1 byte day

The following table describes the contents of the BipSMFRecordHdr record.

Field	Data type	Details
SM117LEN	unsigned short int	SMF record length
SM117SEG	unsigned short int	System reserved
SM117FLG	char	System indicator
SM117RTY	char	Record type 117 (x'75')
SM117TME	unsigned int	Time when SMF moved the record (time since midnight in hundredths of a second)
SM117DTE	unsigned int	Date when SMF moved the record in packed decimal form 0ccyddF where: c is 0 (19xx) or 1 (20xx) yy is the current year (0-99) ddd is the current day (1-366) F is the sign
SM117SID	unsigned int	System ID
SM117SSI	unsigned int	Subsystem ID
SM117STY	unsigned short int	Record subtype, one of : <ul style="list-style-type: none"> • 1 (only message flow or threads data is being collected) • 2 (node data is being collected)¹

Field	Data type	Details
SM117TCT	unsigned int	Count of triplets
SM117SRT	unsigned char	Record type, one of: <ul style="list-style-type: none"> • Archive • Snapshot
SM117SRC	unsigned char	Record code, one of: <ul style="list-style-type: none"> • 00 = None • 01 = Major Interval • 02 = Snapshot • 03 = Shutdown • 04 = Redeploy • 05 = Stats Settings Modified
SM117RSQ	unsigned short int	Sequence number of the record when multiple records are written for a collection interval.
SM117NOR	unsigned short int	Total number of related records in a collection interval.
Note:		
1. When only nodes data is being collected, a single subtype 2 record is written. If nodes and terminals data is being collected, multiple subtype 2 records are written.		

The following table describes the contents of the BipSMFTriplet record.

Field	Data type	Details
TRPLTOSE	signed int	Offset of record from start of SMF record
TRPLTDLE	signed short int	Length of data type
TRPLTNDR	signed short int	Number of data types in SMF record

The following table describes the contents of the BipSMFMessageFlow record.

Field	Data type	Details
IMFLID	short int	Control block hex ID (BipSMFMessageFlow_ID)
IMFLEN	short int	Length of control block
IMFLEYE	char[4]	Eyecatcher (IMFL)
IMFLVER	int	Version number (BipSMFRecordVersion)
IMFLBKNM	char[32]	Broker name
IMFLBKID	char[36]	Broker universal unique identifier
IMFLEXNM	char[32]	Execution group name
IMFLEXID	char[36]	Execution group universal unique identifier
IMFLMFNM	char[32]	Message flow name
IMFLSTDT	BipSMFDate	Interval start date
IMFLSTTM	unsigned int	Interval start time (format as for SM117TME)
IMFLENDT	BipSMFDate	Interval end date
IMFLENTM	unsigned int	Interval end time (format as for SM117TME)
IMFLTPTM	long long int	Total elapsed time spent processing input messages (8 bytes binary, microseconds)
IMFLMXTM	long long int	Maximum elapsed time spent processing an input message (8 bytes binary, microseconds)

Field	Data type	Details
IMFLMNTM	long long int	Minimum elapsed time spent processing an input message (8 bytes binary, microseconds)
IMFLTPCP	long long int	Total CPU time spent processing input messages (8 bytes binary, microseconds)
IMFLMXCP	long long int	Maximum CPU time spent processing an input message (8 bytes binary, microseconds)
IMFLMNCP	long long int	Minimum CPU time spent processing an input message (8 bytes binary, microseconds)
IMFLWTCP	long long int	Total CPU time spent waiting for input messages (8 bytes binary, microseconds)
IMFLWTIN	long long int	Total elapsed time spent waiting for input messages (8 bytes binary, microseconds)
IMFLTPMG	unsigned int	Total number of messages processed
IMFLTSMG	long long int	Total size of input messages (bytes)
IMFLMXMG	long long int	Maximum input message size (bytes)
IMFLMNMG	long long int	Minimum input message size (bytes)
IMFLTHDP	unsigned int	Number of threads in pool
IMFLTHDM	unsigned int	Number of times the maximum number of threads is reached
IMFLERMQ ¹	unsigned int	Number of MQGET errors (MQInput node) or Web services errors (HTTPInput node)
IMFLERMG ²	unsigned int	Number of messages that contain errors
IMFLERPR	unsigned int	Number of errors processing a message
IMFLTMOU	unsigned int	Number of timeouts processing a message (AggregateReply node only)
IMFLCMIT	unsigned int	Number of transaction commits
IMFLBKOU	unsigned int	Number of transaction backouts
IMFLACCT	char[32]	Accounting origin
Notes:		
1. For example, a conversion error occurs when the message is got from the queue.		
2. These include exceptions that are thrown downstream of the input node, and errors detected by the input node after it has successfully retrieved the message from the queue (for example, a format error).		

The following table describes the contents of the BipSMFThread record.

Field	Data type	Details
ITHDID	short int	Control block hex ID (BipSMFThread_ID)
ITHDLEN	short int	Length of control block
ITHDEYE	char[4]	Eyecatcher (ITHD)
ITHDVER	int	Version number (BipSMFRecordVersion)
ITHDNBR	unsigned int	Relative thread number in pool
ITHDTPMG	unsigned int	Total number of messages processed by thread
ITHDTPTM	long long int	Total elapsed time spent processing input messages (8 bytes binary, microseconds)

Field	Data type	Details
ITHDTPCP	long long int	Total CPU time spent processing input messages (8 bytes binary, microseconds)
ITHDWTCP	long long int	Total CPU time spent waiting for input messages (8 bytes binary, microseconds)
ITHDWTIN	long long int	Total elapsed time spent waiting for input messages (8 bytes binary, microseconds)
ITHDTSMG	long long int	Total size of input messages (bytes)
ITHDMXMG	long long int	Maximum size of input messages (bytes)
ITHDMNMG	long long int	Minimum size of input messages (bytes)

The following table describes the contents of the BipSMFNode record.

Field	Data type	Details
INODID	short int	Control block hex ID (BipSMFNode_ID)
INODLEN	short int	Length of control block
INODEYE	char[4]	Eyecatcher (INOD)
INODVER	int	Version number (BipSMFRecordVersion)
INODNDNM	char[32]	Name of node (Label)
INODTYPE	char[32]	Type of node
INODTPTM	long long int	Total elapsed time spent processing input messages (8 bytes binary, microseconds)
INODMXTM	long long int	Maximum elapsed time spent processing input messages (8 bytes binary, microseconds)
INODMNTM	long long int	Minimum elapsed time spent processing input messages (8 bytes binary, microseconds)
INODTPCP	long long int	Total CPU time spent processing input messages (8 bytes binary, microseconds)
INODMXCP	long long int	Maximum CPU time spent processing input messages (8 bytes binary, microseconds)
INODMNCP	long long int	Minimum CPU time spent processing input messages (8 bytes binary, microseconds)
INODTPMG	unsigned int	Total number of messages processed by this node
INODNITL	unsigned int	Number of input terminals
INODNOTL	unsigned int	Number of output terminals

The following table describes the contents of the BipSMFTerminal record.

Field	Data type	Details
ITRMID	short int	Control block hex ID (BipSMFTerminal_ID)
ITRMLN	short int	Length of control block
ITRMEYE	char[4]	Eyecatcher (ITRM)
ITRMVER	int	Version number (BipSMFRecordVersion)
ITRMTLNM	char[32]	Name of terminal

Field	Data type	Details
ITRMTYPE	char[8]	Type of terminal, one of: <ul style="list-style-type: none"> • Input • Output
ITRMTINV	unsigned int	Total number of invocations

Example message flow accounting and statistics data

The following topics give example output in two formats:

- XML publication
- User trace entries

An example is not provided for z/OS SMF records, because these contain hexadecimal data and are not easily viewed in that form. To view SMF records, use any available utility program that processes SMF records. For example, you can download WebSphere MQ SupportPac IS11, which generates formatted SMF records that are very similar to formatted user trace entries.

Example of an XML publication for message flow accounting and statistics

This topic shows an XML publication that contains message flow accounting and statistics data.

The following example shows the output generated for a snapshot report. The content of this publication message shows that the message flow is called XMLflow, and that it is running in an execution group named default on broker MQ02BRK. The message flow contains the following nodes:

- An MQInput node called INQueue3
- An MQOutput node called OUTQueue
- An MQOutput node called FAILQueue

The MQInput node out terminal is connected to the OUTQueue node. The MQInput node failure terminal is connected to the FAILQueue node.

During the interval for which statistics have been collected, this message flow processed no messages.

A publication generated for this data always includes the appropriate folders, even if there is no current data.

The following command has been issued to achieve these results:

```
mqsichangeflowstats MQ02BRK -s -c active -e default -f XMLFlow -n advanced -t basic -b basic -o xml
```

Blank lines have been added between folders to improve readability.

The following example is the subscription message. The <psc> and <mcd> elements are part of the RFH header.

```
<psc>
  <Command>Publish</Command>
  <PubOpt>RetainPub</PubOpt>
  <Topic>${SYS}/Broker/MQ02BRK/StatisticsAccounting/SnapShot/default/XMLflow
</Topic>
</psc>
```

```
<mcd>
  <Msd>xml</Msd>
</mcd>
```

The following example is the publication that the broker generates:

```
<WMQIStatisticsAccounting RecordType="Snapshot" RecordCode="Snapshot">

<MessageFlow BrokerLabel="MQ02BRK"
  BrokerUUID="7d951e31-f200-0000-0080-efe1b9d849dc"
  ExecutionGroupName="default"
  ExecutionGroupUUID="77cf1e31-f200-0000-0080-efe1b9d849dc"
  MessageFlowName="XMLflow" StartDate="2003-01-17"
  StartTime="14:44:34.581320" EndDate="2003-01-17" EndTime="14:44:44.582926"
  TotalElapsedTime="0"
  MaximumElapsedTime="0" MinimumElapsedTime="0" TotalCPUTime="0"
  MaximumCPUTime="0" MinimumCPUTime="0" CPUTimeWaitingForInputMessage="685"
  ElapsedTimeWaitingForInputMessage="10001425" TotalInputMessages="0"
  TotalSizeOfInputMessages="0" MaximumSizeOfInputMessages="0"
  MinimumSizeOfInputMessages="0" NumberOfThreadsInPool="1"
  TimesMaximumNumberOfThreadsReached="0" TotalNumberOfMQErrors="0"
  TotalNumberOfMessagesWithErrors="0" TotalNumberOfErrorsProcessingMessages="0"
  TotalNumberOfTimeOutsWaitingForRepliesToAggregateMessages="0"
  TotalNumberOfCommits="0" TotalNumberOfBackouts="0" AccountingOrigin="DEPT1"/>

<Threads Number="1">
<ThreadStatistics Number="5" TotalNumberOfInputMessages="0"
  TotalElapsedTime="0" TotalCPUTime="0" CPUTimeWaitingForInputMessage="685"
  ElapsedTimeWaitingForInputMessage="10001425" TotalSizeOfInputMessages="0"
  MaximumSizeOfInputMessages="0" MinimumSizeOfInputMessages="0"/>
</Threads>

<Nodes Number="3">

  <NodeStatistics Label="FAILQueue" Type="MQOutput" TotalElapsedTime="0"
    MaximumElapsedTime="0" MinimumElapsedTime="0" TotalCPUTime="0"
    MaximumCPUTime="0" MinimumCPUTime="0" CountOfInvocations="0"
    NumberOfInputTerminals="1" NumberOfOutputTerminals="2">
    <TerminalStatistics Label="failure" Type="Output" CountOfInvocations="0"/>
    <TerminalStatistics Label="in" Type="Input" CountOfInvocations="0"/>
    <TerminalStatistics Label="out" Type="Output" CountOfInvocations="0"/>
  </NodeStatistics>

  <NodeStatistics Label="INQueue3" Type="MQInput" TotalElapsedTime="0"
    MaximumElapsedTime="0" MinimumElapsedTime="0" TotalCPUTime="0"
    MaximumCPUTime="0" MinimumCPUTime="0" CountOfInvocations="0"
    NumberOfInputTerminals="0" NumberOfOutputTerminals="3">
    <TerminalStatistics Label="catch" Type="Output" CountOfInvocations="0"/>
    <TerminalStatistics Label="failure" Type="Output" CountOfInvocations="0"/>
    <TerminalStatistics Label="out" Type="Output" CountOfInvocations="0"/>
  </NodeStatistics>

  <NodeStatistics Label="OUTQueue" Type="MQOutput" TotalElapsedTime="0"
    MaximumElapsedTime="0" MinimumElapsedTime="0" TotalCPUTime="0"
    MaximumCPUTime="0" MinimumCPUTime="0" CountOfInvocations="0"
    NumberOfInputTerminals="1" NumberOfOutputTerminals="2">
    <TerminalStatistics Label="failure" Type="Output" CountOfInvocations="0"/>
    <TerminalStatistics Label="in" Type="Input" CountOfInvocations="0"/>
    <TerminalStatistics Label="out" Type="Output" CountOfInvocations="0"/>
  </NodeStatistics>
```

</Nodes>

</WMQIStatisticsAccounting>

Example of user trace entries for message flow accounting and statistics

This topic shows a user trace that contains message flow accounting and statistics data.

The following example shows the output generated for a snapshot report. The messages written to the trace show that the message flow is called `myExampleFlow`, and that it is running in an execution group named `default` on broker `MQ01BRK`. The message flow contains the following nodes:

- An MQInput node called `inNode`
- A Compute node called `First1`
- An MQOutput node called `outNode`

The nodes are connected together (out terminal to in terminal for each connection).

During the interval for which statistics have been collected, this message flow processed 150 input messages.

The records show that there are two threads assigned to this message flow. One thread is assigned when the message flow is deployed (this is the default number); an additional thread (thread 0) listens on the input queue. The listening thread starts additional threads to process input messages dependent on the number of instances that you have configured for the message flow, and the rate of arrival of the input messages on the input queue.

The following command has been issued to achieve these results:

```
mqsichangeflowstats MQ01BRK -s -c active -e default -f myExampleFlow -n advanced -t basic -b basic
```

The trace entries have been retrieved with the `mqsireadlog` command and formatted using the `mqsiformatlog` command. The output from `mqsiformatlog` is shown below. Line breaks have been added to aid readability.

```
BIP2380I: WMQI message flow statistics. ProcessID='328467', Key='6', Type='SnapShot', Reason='Snapshot',
BrokerLabel='MQ01BRK', BrokerUUID='18792e66-e100-0000-0080-f197e5ed81bd',
ExecutionGroupName='default', ExecutionGroupUUID='15d4314a-3607-11d4-8000-09140f7b0000',
MessageFlowName='myExampleFlow',
StartDate='2003-05-20', StartTime='13:44:31.885862',
EndDate='2003-05-20', EndTime='13:44:51.310080',
TotalElapsedTime='9414843', MaximumElapsedTime='1143442', MinimumElapsedTime='35154',
TotalCPUTime='760147', MaximumCPUTime='70729', MinimumCPUTime='3124',
CPUTimeWaitingForInputMessage='45501', ElapsedTimeWaitingForInputMessage='11106438',
TotalInputMessages='150', TotalSizeOfInputMessages='437250',
MaximumSizeOfInputMessages='2915', MinimumSizeOfInputMessages='2915',
NumberOfThreadsInPool='1', TimesMaximumNumberOfThreadsReached='150',
TotalNumberOfMQErrors='0', TotalNumberOfMessagesWithErrors='0',
TotalNumberOfErrorsProcessingMessages='0', TotalNumberOfTimeOuts='0',
TotalNumberOfCommits='150', TotalNumberOfBackouts='0', AccountingOrigin="DEPT2".
Statistical information for message flow 'myExampleFlow' in broker 'MQ01BRK'.
This is an information message produced by WMQI statistics.
```

```
BIP2381I: WMQI thread statistics. ProcessID='328467', Key='6', Number='0',
TotalNumberOfInputMessages='0',
TotalElapsedTime='0', TotalCPUTime='0', CPUTimeWaitingForInputMessage='110',
ElapsedTimeWaitingForInputMessage='5000529', TotalSizeOfInputMessages='0',
```

MaximumSizeOfInputMessages='0', MinimumSizeOfInputMessages='0'.
Statistical information for thread '0'.
This is an information message produced by WMQI statistics.

BIP2381I: WMQI thread statistics. ProcessID='328467', Key='6', Number='18',
TotalNumberOfInputMessages='150',
TotalElapsedTime='9414843', TotalCPUTime='760147', CPUTimeWaitingForInputMessage='45391',
ElapsedTimeWaitingForInputMessage='6105909', TotalSizeOfInputMessages='437250',
MaximumSizeOfInputMessages='2915', MinimumSizeOfInputMessages='2915'.
Statistical information for thread '18'.
This is an information message produced by WMQI statistics.

BIP2382I: WMQI node statistics. ProcessID='328467', Key='6',
Label='First1', Type='ComputeNode',
TotalElapsedTime='6428815', MaximumElapsedTime='138261', MinimumElapsedTime='28367',
TotalCPUTime='604060', MaximumCPUTime='69645', MinimumCPUTime='2115',
CountOfInvocations='150', NumberOfInputTerminals='1', NumberOfOutputTerminals='2'.
Statistical information for node 'First1'.
This is an information message produced by WMQI statistics.

BIP2383I: WMQI terminal statistics. ProcessID='328467', Key='6',
Label='failure', Type='Output', CountOfInvocations='0',
Statistical information for terminal 'failure'.
This is an information message produced by WMQI statistics.

BIP2383I: WMQI terminal statistics. ProcessID='328467', Key='6',
Label='in', Type='Input', CountOfInvocations='150',
Statistical information for terminal 'in'.
This is an information message produced by WMQI statistics.

BIP2383I: WMQI terminal statistics. ProcessID='328467', Key='6',
Label='out', Type='Output', CountOfInvocations='150',
Statistical information for terminal 'out'.
This is an information message produced by WMQI statistics.

BIP2382I: WMQI node statistics. ProcessID='328467', Key='6',
Label='inNode', Type='MQInputNode',
TotalElapsedTime='1813446', MaximumElapsedTime='1040209', MinimumElapsedTime='1767',
TotalCPUTime='70565', MaximumCPUTime='686', MinimumCPUTime='451',
CountOfInvocations='150', NumberOfInputTerminals='0', NumberOfOutputTerminals='3'.
Statistical information for node 'inNode'.
This is an information message produced by WMQI statistics.

BIP2383I: WMQI terminal statistics. ProcessID='328467', Key='6',
Label='catch', Type='Output', CountOfInvocations='0',
Statistical information for terminal 'catch'.
This is an information message produced by WMQI statistics.

BIP2383I: WMQI terminal statistics. ProcessID='328467', Key='6',
Label='failure', Type='Output', CountOfInvocations='0',
Statistical information for terminal 'failure'.
This is an information message produced by WMQI statistics.

BIP2383I: WMQI terminal statistics. ProcessID='328467', Key='6',
Label='out', Type='Output', CountOfInvocations='150',
Statistical information for terminal 'out'.
This is an information message produced by WMQI statistics.

BIP2382I: WMQI node statistics. ProcessID='328467', Key='6',
Label='outNode', Type='MQOutputNode',
TotalElapsedTime='1172582', MaximumElapsedTime='177516', MinimumElapsedTime='3339',
TotalCPUTime='85522', MaximumCPUTime='762', MinimumCPUTime='536',
CountOfInvocations='150', NumberOfInputTerminals='1', NumberOfOutputTerminals='2'.
Statistical information for node 'outNode'.
This is an information message produced by WMQI statistics.

BIP2383I: WMQI terminal statistics. ProcessID='328467', Key='6',
Label='failure', Type='Output', CountOfInvocations='0',

Statistical information for terminal 'failure'.
This is an information message produced by WMQI statistics.

BIP2383I: WMQI terminal statistics. ProcessID='328467', Key='6',
Label='in', Type='Input', CountOfInvocations='150',
Statistical information for terminal 'in'.
This is an information message produced by WMQI statistics.

BIP2383I: WMQI terminal statistics. ProcessID='328467', Key='6',
Label='out', Type='Output', CountOfInvocations='0',
Statistical information for terminal 'out'.
This is an information message produced by WMQI statistics.

Coordinated message flows

The following topics provide reference information for database use in coordinated message flows:

- “Database connections for coordinated message flows”
- “Database support for coordinated message flows”

Database connections for coordinated message flows

When you configure a message flow to access a database, the broker establishes a connection to that database based on the ODBC DSN. To coordinate the database updates with other updates (determined by the configuration you have set for each node that accesses a database), the broker makes a connection for each transaction mode for each DSN accessed on each message flow thread.

Therefore if you set the *Transaction Mode* property for one node in the message flow to *Automatic* and for another node to *Commit*, the broker establishes two separate connections to this DSN from the same thread. Take this into account when you calculate the number of connections required between a broker and a specific DSN.

For further information about connections made by the broker to user databases, see “User database connections” on page 700.

Connections to user databases are in addition to the runtime connections that are required by the broker (to the DB2, Oracle, Sybase, or SQL Server database that is defined to hold its internal information). For details of these connections, refer to *Configuring access to databases*.

Database support for coordinated message flows

If the message flow processing includes interaction with an external database, the transaction can be coordinated using XA technology.

This ensures that all participants update or return to a consistent state. This external coordination support is provided by the underlying WebSphere MQ facilities on distributed systems, and by Resource Recovery Services (RRS) on z/OS.

The following databases provide the correct level of XA support for coordinating message flows on distributed systems:

- DB2

Globally coordinated message flows that involve a DB2 resource manager are supported on DB2 Universal Database V8.

- Oracle
- Sybase

On z/OS, database support for coordinated message flows is provided by DB2 only.

Element definitions for message parsers

The topics in this section discuss data types for the WebSphere MQ headers, and define the element names, types, and attributes for each of the supported headers:

- “Data types of fields and elements”
- “The MQCFH parser” on page 741
- “The MQCIH parser” on page 742
- “The MQDLH parser” on page 743
- “The MQIIH parser” on page 743
- “The MQMD parser” on page 744
- “The MQMDE parser” on page 745
- “The MQRFH parser” on page 745
- “The MQRFH2 parser” on page 746
- “The MQRMH parser” on page 746
- “The MQSAPH parser” on page 747
- “The MQWIH parser” on page 747
- “The SMQ_BMH parser” on page 747

For each parser, the following terms are defined:

- Root element name: the name of the syntax element created by the parser at the root of its own part of the tree.
- Class name: the name by which the parser defines itself to WebSphere Message Broker.

Data types of fields and elements

The fields within WebSphere MQ headers and other subtrees built from the message are of a particular data type. When you manipulate the messages and their headers using ESQL in the message flow nodes, be aware of type information in field references:

- “Data types of the fields in the WebSphere MQ headers”
- “Data types for elements in the Properties subtree” on page 736
- “Data types for elements in the DestinationData subtree” on page 736
- “Data types for elements in an MRM message” on page 738
- “Data types for an unstructured (BLOB) message” on page 739
- “Field names of the IDoc parser structures” on page 740

Data types of the fields in the WebSphere MQ headers

The fields in the WebSphere MQ headers have specific data types. Parsers are supplied for the WebSphere MQ headers listed below. The parsers determine the data type of each field in the header:

- “The MQCFH parser” on page 741
- “The MQCIH parser” on page 742
- “The MQDLH parser” on page 743
- “The MQIIH parser” on page 743

- “The MQMD parser” on page 744
- “The MQMDE parser” on page 745
- “The MQRFH parser” on page 745
- “The MQRFH2 parser” on page 746
- “The MQRMH parser” on page 746
- “The MQSAPH parser” on page 747
- “The MQWIH parser” on page 747
- “The SMQ_BMH parser” on page 747

The mapping of the WebSphere MQ data types to the data types used in the broker is shown in the table below:

Data type of the field	Represented as
MQLONG	INTEGER
MQCHAR, MQCHAR4	CHARACTER
MQBYTE, MQBYTE _n	BLOB

Data types for elements in the Properties subtree

A parser is supplied for the Properties subtree; it associates each field with a specific data type.

The fields and data type of each field are shown in the table below:

Data type of the element	Represented as
CodedCharSetId	INTEGER
CreationTime	TIMESTAMP
ContentType	CHARACTER
Encoding	INTEGER
ExpirationTime	TIMESTAMP
MessageFormat	CHARACTER
MessageSet	CHARACTER
MessageType	CHARACTER
Persistence	BOOLEAN
Priority	INTEGER
ReplyIdentifier	CHARACTER
ReplyProtocol	CHARACTER
Topic (this field contains a list)	CHARACTER
Transactional	BOOLEAN

Data types for elements in the DestinationData subtree

The DestinationData subtree is part of the Destination subtree in the LocalEnvironment. LocalEnvironment trees are created by input nodes when they receive a message and, optionally, by compute nodes. When created, they are empty but you can create data in them by using ESQL statements coded in any of the SQL nodes.

The Destination subtree consists of subtrees for zero or more protocols, for example WebSphere MQ and WebSphere MQ Everyplace, or a subtree for routing destinations (RouterList), or both.

The protocol tree has two children:

- Defaults is the first element. There can be only one.
- DestinationData is the following element, and can be repeated any number of times, to represent each destination to which a message is sent.

“LocalEnvironment tree” on page 18 includes a picture of a typical tree, showing a Destination tree that has both protocol and RouterList subtrees.

The structure of data within the DestinationData folder is the same as that in Defaults for the same protocol, and can be used to override the default values in Defaults. You can therefore set up Defaults to contain values that are common to all destinations, and set only the unique values in each DestinationData subtree. If a value is set neither in DestinationData, nor in Defaults, the value that you have set for the corresponding node property is used.

The fields, data type, and valid values for each element of Defaults and DestinationData subtrees for WebSphere MQ are shown in the following table. “MQOutput node” on page 605 describes the corresponding node properties.

Refer to “Accessing the LocalEnvironment tree” on page 192 for information about using DestinationData.

Data type of the element	Represented as	Corresponding node property	Valid values
queueManagerName	CHARACTER	Queue Manager Name	
queueName	CHARACTER	Queue Name	
transactionMode	CHARACTER	Transaction Mode	no, yes, automatic
persistenceMode	CHARACTER	Persistence Mode	no, yes, automatic, asQdef
newMsgId	CHARACTER	New Message ID	no, yes
newCorrelId	CHARACTER	New Correlation ID	no, yes
segmentationAllowed	CHARACTER	Segmentation Allowed	no, yes
alternateUserAuthority	CHARACTER	Alternate User Authority	no, yes
replyToQMgr	CHARACTER	Reply-to queue manager	
replyToQ	CHARACTER	Reply-to queue	

Case-sensitivity for data types and values

When you create these fields in the DestinationData folder, you need to enter the data type and value exactly as shown in the table . If any variations in spelling or case are used then these fields or values are ignored in the DestinationData records and the next available value is used.

For example, the following ESQL samples could result in unexpected output:

```
SET OutputLocalEnvironment.Destination.MQ.DestinationData[1].persistenceMode = 'YES';
SET OutputLocalEnvironment.Destination.MQ.DestinationData[2].PersistenceMode = 'yes';
```

In each case the DestinationData folder might not write a persistent message for these destinations. In the first example the persistenceMode field has been given a value of 'YES', which is not one of the valid values listed in the table above and this value is ignored. In the second example, the field named 'PersistenceMode' is specified incorrectly and is ignored. Either the persistenceMode value of the Defaults folder, or the value of the associated attribute on the MQOutput node will be used. If this causes a value of 'no' or 'automatic' to be used, a persistent message will not be written.

If a DestinationData folder is producing unexpected output, you should check that you have used the correct case and spelling in the fields and values used.

Data types for elements in an MRM message

A parser is supplied for the body of a message in the MRM domain; it associates each field with a specific data type.

The following table shows the mapping from XML Schema data types that you have specified for elements in the MRM to data types used by the broker and supported by ESQL. When you create an element, you might find that associated value constraints are created to ensure a more accurate mapping of the XML Schema type.

Data type of the element	Represented as
ANYURI	STRING
BASE64BIN	BINARY
BOOLEAN	BOOLEAN
BYTE	INTEGER
DATE	DATETIME
DATETIME	DATETIME
DECIMAL	DECIMAL
DOUBLE	FLOAT
DURATION	STRING
ENTITIES	STRING
ENTITY	STRING
FLOAT	FLOAT
GDAY	DATETIME
GMONTH	DATETIME
GMONTHDAY	DATETIME
GYEAR	DATETIME
GYEARMONTH	DATETIME
HEXBINARY	BINARY
ID	STRING
IDREF	STRING
IDREFS	STRING
INT	INTEGER

Data type of the element	Represented as
INTEGER	DECIMAL
LANGUAGE	STRING
LONG	INTEGER
NAME	STRING
NCNAME	STRING
NEGATIVE_INTEGER	DECIMAL
NMTOKEN	STRING
NMTOKENS	STRING
NON_NEGATIVE_INT	DECIMAL
NON_POSITIVE_INTEGER	DECIMAL
NORMALIZED_STRING	STRING
NOTATION	STRING
POSITIVE_INTEGER	DECIMAL
QNAME	STRING
SHORT	INTEGER
STRING	STRING
TIME	DATETIME
TOKEN	STRING
UNSIGNED_BYTE	INTEGER
UNSIGNEDINT	INTEGER
UNSIGNEDLONG	INTEGER
UNSIGNED_SHORT	INTEGER

Simple type - list

In the message tree, a list type will be represented as a name node with an anonymous value child for each list item. This allows repeating lists to be handled without any loss of information. Repeating lists will appear as sibling name elements, each of which has its own anonymous value child nodes for its respective list items.

Data types for an unstructured (BLOB) message

A parser is supplied for the body of a message in the BLOB domain; it associates each field with a specific data type.

An unstructured (BLOB) message has the data types shown below:

Data type of the element	Represented as
BLOB	BLOB
UnknownParserName	CHARACTER

The UnknownParserName field, if present, contains the class name of the parser chosen in preference to the BLOB parser. This information is used by the header integrity routine (described in “Parsers” on page 26) to ensure that the semantic meaning of the message is preserved.

Field names of the IDoc parser structures

This topic lists all the field names of the Control Structure (DC) and the Data Structure (DD) used by the IDoc parser. They are documented in the form that they are used in a SET statement of ESQL.

For example:

```
SET OutputRoot.Properties = InputRoot.Properties;
SET OutputRoot.MQMD = InputRoot.MQMD;
```

Control structure (DC) fields:

All the fields must be specified and set.

The syntax is:

```
<rootname>.<ParserName>.<foldername>.<fieldname>=
```

For example:

```
SET "OutputRoot"."IDOC"."DC"."docnum" = '0000000000000001';
SET "OutputRoot"."IDOC"."DC"."idoctyp" = 'MATMAS01'
```

The field names, which must be specified in order, are:

1) tabnam	2) mandt	3) docnum
4) docrel	5) status	6) direct
7) outmod	8) exprss	9) test
10) idoctyp	11) cimtyp	12) mestyp
13) mescod	14) mesfct	15) std
16) stdvrs	17) stdmes	18) sndpor
19) sndprt	20) sndpfc	21) sndprn
22) sndsad	23) sndlad	24) rcvpor
25) rcvprt	26) rcvpfc	27) rcvprn
28) rcvsad	29) rcvlad	30) credat
31) cretim	32) refint	33) refgrp
34) refmes	35) arckey	36) serial

Data structure (DD) fields:

To access each DD segment, use the array suffix as follows: DD[1], DD[2], and so on.

The syntax is:

```
<rootname>.<ParserName>.DD[1].<fieldname>=
```

For example:

```
SET OutputRoot.IDOC.DD[I].segnam = 'E2MAKTM001';
SET OutputRoot.IDOC.DD[I].mandt2 = '111';
```

In the following table, note the use of the suffix 2 to give unique field names to the mandt and docnum fields.

The field names, which must be supplied in order, are:

1) segnam	2) mandt2	3) docnum2
4) segnum	5) psgnum	6) hlevel

Note:

1. The last 1000 bytes of data in the DD segment are the bytes modelled in the MRM.
2. The DD segnam describes which model the MRM uses.

Segment fields:

The syntax is:

<rootname>.<ParserName>.DD[1].sdatatag.MRM.<fieldname>=

For example:

```
SET OutputRoot.IDOC.DD[I].sdatatag.MRM.msgfn = '006'
SET OutputRoot.IDOC.DD[I].sdatatag.MRM.spras_iso = 'EN'
```

Note:

1. The sdatatag keyword indicates to the parser that it is the element containing the data to be manipulated
2. The MRM keyword indicates that the MRM handles the transformation

msgfn	spras	maktx
msgfn	spras_iso	fill954

The fill954 keyword is the filler for the segment, because an incoming IDoc to SAP must have 1000 byte segments

The MQCFH parser

The root name for this parser is MQPCF. The class name is MQPCF.

The table below lists the elements native to the MQCFH header:

Element Name	Element Data Type	Element Attributes
Type	INTEGER	Name Value
StrucLength	INTEGER	Name Value
Version	INTEGER	Name Value
Command	INTEGER	Name Value
MsgSeqNumber	INTEGER	Name Value
Control	INTEGER	Name Value
CompCode	INTEGER	Name Value
Reason	INTEGER	Name Value
ParameterCount	INTEGER	Name Value

For further information about this header and its contents, see the *WebSphere MQ Programmable Command Formats and Administration Interface* book.

The MQCIH parser

The root name for this parser is MQCIH. The class name is MQCICS.

The table below lists the elements native to the MQCIH header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
ReturnCode	INTEGER	Name Value
CompCode	INTEGER	Name Value
Reason	INTEGER	Name Value
UOWControl	INTEGER	Name Value
GetWaitInterval	INTEGER	Name Value
LinkType	INTEGER	Name Value
OutputDataLength	INTEGER	Name Value
FacilityKeepTime	INTEGER	Name Value
ADSDescriptor	INTEGER	Name Value
ConversationalTask	INTEGER	Name Value
TaskEndStatus	INTEGER	Name Value
Facility	BLOB	Name Value
Function	CHARACTER	Name Value
AbendCode	CHARACTER	Name Value
Authenticator	CHARACTER	Name Value
Reserved1	CHARACTER	Name Value
ReplyToFormat	CHARACTER	Name Value
RemoteSysId	CHARACTER	Name Value
RemoteTransId	CHARACTER	Name Value
TransactionId	CHARACTER	Name Value
FacilityLike	CHARACTER	Name Value
AttentionId	CHARACTER	Name Value
StartCode	CHARACTER	Name Value
CancelCode	CHARACTER	Name Value
NextTransactionId	CHARACTER	Name Value
Reserved2	CHARACTER	Name Value
Reserved3	CHARACTER	Name Value
CursorPosition	INTEGER	Name Value
ErrorOffset	INTEGER	Name Value

Element Name	Element Data Type	Element Attributes
InputItem	INTEGER	Name Value
Reserved4	INTEGER	Name Value

The MQDLH parser

The root name for this parser is MQDLH. The class name is MQDEAD.

The table below lists the elements native to the MQDLH header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Reason	INTEGER	Name Value
DestQName	CHARACTER	Name Value
DestQMgrName	CHARACTER	Name Value
PutApplType	INTEGER	Name Value
PutApplName	CHARACTER	Name Value
PutDate	TIMESTAMP/CHARACTER	Name Value
PutTime	TIMESTAMP/CHARACTER	Name Value

The MQIIH parser

The root name for this parser is MQIIH. The class name is MQIMS.

The table below lists the elements native to the MQIIH header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
LTermOverride	CHARACTER	Name Value
MFSMapName	CHARACTER	Name Value
ReplyToFormat	CHARACTER	Name Value
Authenticator	CHARACTER	Name Value
TranInstanceId	BLOB	Name Value
TranState	CHARACTER	Name Value
CommitMode	CHARACTER	Name Value
SecurityScope	CHARACTER	Name Value
Reserved	CHARACTER	Name Value

The MQMD parser

The root name for this parser is MQMD. The class name is MQHMD.

The table below lists the orphan elements adopted by the MQMD header:

Element Name	Element Data Type	Element Attributes
SourceQueue	CHARACTER	Name Value
Transactional	BOOLEAN	Name Value

The table below lists the elements native to the MQMD header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Report	INTEGER	Name Value
MsgType	INTEGER	Name Value
Expiry ¹	INTEGER/GMTTIMESTAMP	Name Value
Feedback	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Priority	INTEGER	Name Value
Persistence	INTEGER	Name Value
MsgId	BLOB	Name Value
CorrelId	BLOB	Name Value
BackoutCount	INTEGER	Name Value
ReplyToQ	CHARACTER	Name Value
ReplyToQMgr	CHARACTER	Name Value
UserIdentifier	CHARACTER	Name Value
AccountingToken	BLOB	Name Value
AppIdentityData	CHARACTER	Name Value
PutApplType	INTEGER	Name Value
PutApplName	CHARACTER	Name Value
PutDate	TIMESTAMP/CHARACTER	Name Value
PutTime	TIMESTAMP/CHARACTER	Name Value
AppOriginData	CHARACTER	Name Value
GroupId	BLOB	Name Value
MsgSeqNumber	INTEGER	Name Value
Offset	INTEGER	Name Value
MsgFlags	INTEGER	Name Value
OriginalLength	INTEGER	Name Value

Element Name	Element Data Type	Element Attributes
Note:		
1. The Expiry field in the MQMD is a special case:		
<ul style="list-style-type: none"> • An INTEGER value represents an expiry interval in tenths of a second. If the Expiry field is set to -1, it represents an unlimited expiry interval (that is, the message never expires) If the Expiry field is a positive INTEGER, it represents an expiry interval of that number of tenths of a second (for example, if it is set to 4, it represents 4 tenths of a second, if it is set to 15, it represents one and a half seconds). • A GMTTIMESTAMP value represents a specific expiration time. 		
If Expiry contains a GMTTIMESTAMP in the past, or an INTEGER of less than 1 (excluding -1), it is set to the value 1 (one tenth of a second, the minimum value).		

The MQMDE parser

The root name for this parser is MQMDE. The class name is MQHMDE.

The table below lists the elements native to the MQMDE header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
GroupId	BLOB	Name Value
MsgSeqNumber	INTEGER	Name Value
Offset	INTEGER	Name Value
MsgFlags	INTEGER	Name Value
OriginalLength	INTEGER	Name Value

The MQRFH parser

The root name for this parser is MQRFH. The class name is MQHRE.

The table below lists the elements native to the MQRFH header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value

Other name value elements might be present that contain information as parsed from or destined for the option buffer. The *MQSeries Publish/Subscribe User's Guide* provides further information about the MQRFH header.

The MQRFH2 parser

The root name for this parser is MQRFH2. The class name is MQHREF2.

The table below lists the elements native to the MQRFH2 header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
NameValueCCSID	INTEGER	Name Value

Other name and child name value elements might be present that contain information as parsed from or destined for the option buffer. See MQRFH2 header for further information about this header.

The MQRMH parser

The root name for this parser is MQRMH. The class name is MQHREF.

The table below lists the elements native to the MQRMH header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
ObjectType	CHARACTER	Name Value
ObjectInstanceId	BLOB	Name Value
SrcEnv	CHARACTER ¹	Name Value
SrcName	CHARACTER ²	Name Value
DestEnv	CHARACTER ³	Name Value
DestName	CHARACTER ⁴	Name Value
DataLogicalLength	INTEGER	Name Value
DataLogicalOffset	INTEGER	Name Value
DataLogicalOffset2	INTEGER	Name Value
Notes: 1. This field represents both SrcEnvLength and Offset 2. This field represents both SrcNameLength and Offset 3. This field represents both DestEnvLength and Offset 4. This field represents both DestNameLength and Offset		

The MQSAPH parser

The root name for this parser is MQSAPH. The class name is MQHSAP.

The table below lists the elements native to the MQSAPH header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
Client	CHARACTER	Name Value
Language	CHARACTER	Name Value
HostName	CHARACTER	Name Value
UserId	CHARACTER	Name Value
Password	CHARACTER	Name Value
SystemNumber	CHARACTER	Name Value
Reserved	BLOB	Name Value

The MQWIH parser

The root name for this parser is MQWIH. The class name is MQHWIH.

The table below lists the elements native to the MQWIH header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
Flags	INTEGER	Name Value
ServiceName	CHARACTER	Name Value
ServiceStep	CHARACTER	Name Value
MsgToken	BLOB	Name Value
Reserved	CHARACTER	Name Value

The SMQ_BMH parser

The root name for this parser is SMQ_BMH. The class name is SMQBAD.

The table below lists the elements native to the SMQ_BMH header:

Element Name	Element Data Type	Element Attributes
Format	CHARACTER	Name Value
Version	INTEGER	Name Value

Element Name	Element Data Type	Element Attributes
Encoding	INTEGER	Name Value
CodedCharSetId	INTEGER	Name Value
ErrorType	INTEGER	Name Value
Reason	INTEGER	Name Value
PutApplType	INTEGER	Name Value
PutApplName	CHARACTER	Name Value
PutDate	TIMESTAMP/CHARACTER	Name Value
PutTime	TIMESTAMP/CHARACTER	Name Value

Message mappings

This section contains topics that provide reference information about message mapping:

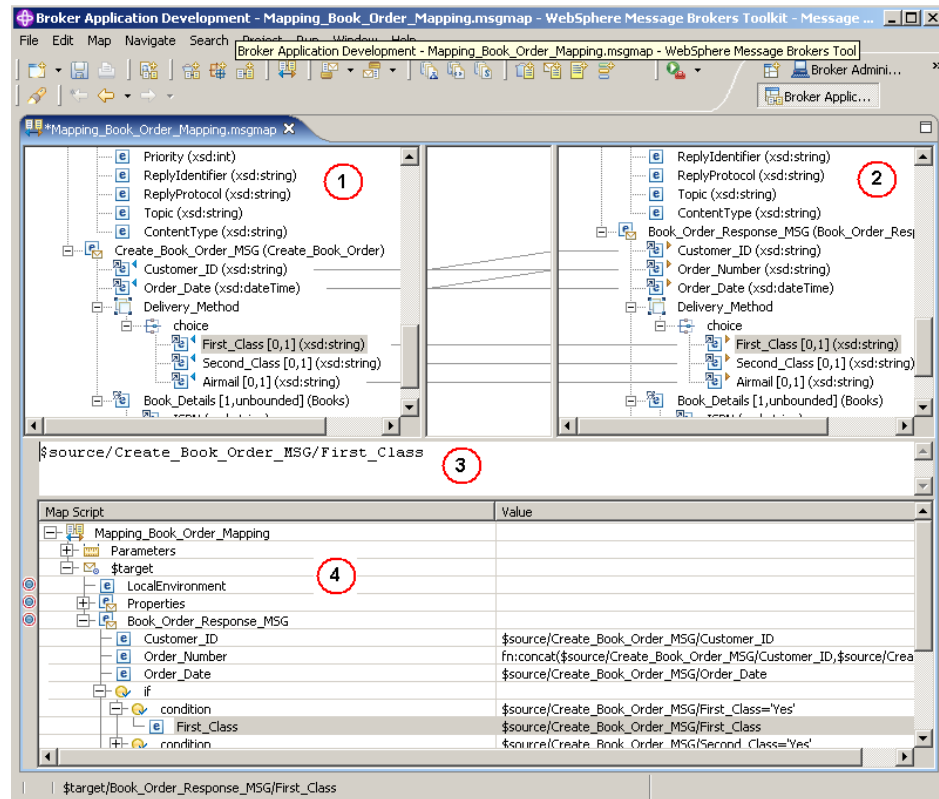
- “Message Mapping editor”
 - Source pane
 - Target pane
 - Edit pane
 - Spreadsheet pane
- “Mapping node” on page 756
 - Syntax
 - Functions
 - Casts
- “Migration of message mappings from Version 5.0” on page 759
 - Migration restrictions

Message Mapping editor

You configure a message mapping using the Message Mapping editor, which you use to set values for:

- the message destination
- message headers
- message content

Here is an example of the Message Mapping editor. There are separate panes for working with sources, targets and expressions, as well as a spreadsheet view.



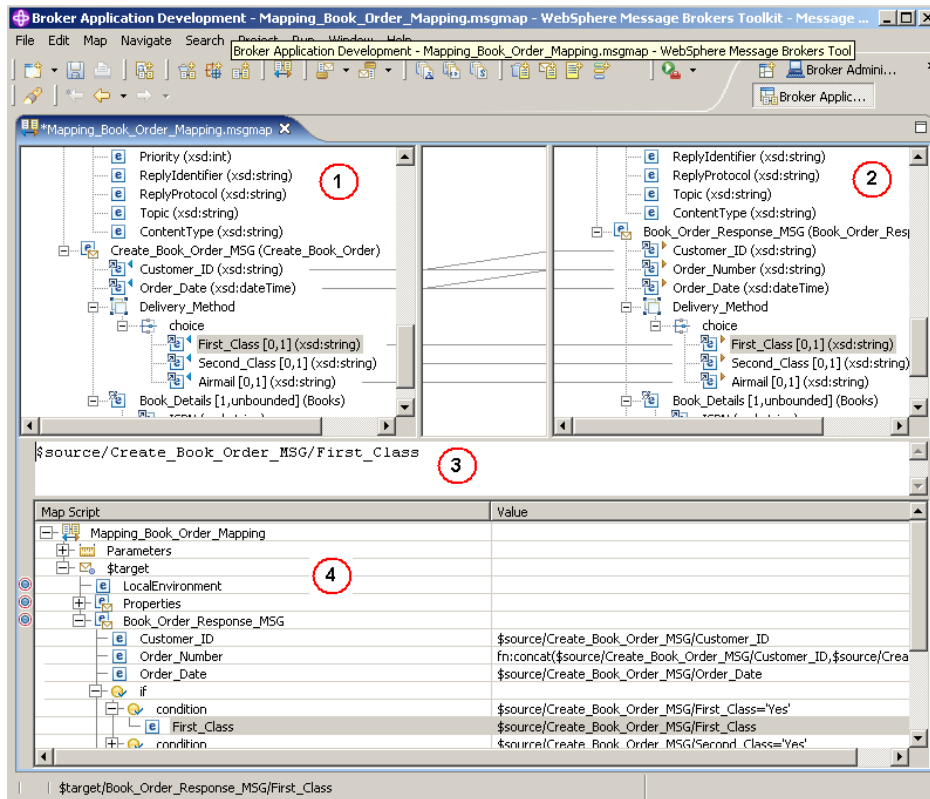
1. **Source pane:** displays a source message or database table
2. **Target pane:** displays a target message
3. **Edit pane:** displays the expression to be used to derive the target element value
4. **Spreadsheet pane:** displays a summary of the mappings in spreadsheet columns (each target field and its value)

Use the Message Mapping editor to perform various mapping tasks.

Wizards and dialog boxes are provided for tasks such as adding mappable elements, working with ESQL, and working with submaps. Mappings that are created with the Message Mapping editor are automatically validated and compiled, ready for adding to a broker archive (bar) file, and subsequent deployment to WebSphere Message Broker.

Message Mapping editor Source pane

The following example shows the “Message Mapping editor” on page 748. The pane that is labelled as 1 in the example is the Source pane:



The following list describes the elements that are present in the Source pane:

- A source message is identified by \$source.
- A source database is identified by \$db:select.
- A mapped entry is indicated by a blue triangle alongside the element. In this example, Customer_ID and Order_Date are mapped.
- Square brackets contain minimum and maximum occurrences of an element.
- An optional field is indicated by [0,1]. In this example, First_Class is optional.
- A repeating field is indicated by [minoccurs, maxoccurs].
- A choice field is indicated by a choice line; under the choice line are the possible choices. In this example, First_Class, Second_Class, and Airmail are choices of Delivery_Method.
- The type of each element is indicated in round brackets after the element name.
- If the message schema uses namespaces, the namespace prefix is shown before the element name, separated by a colon.

Use the Source pane to invoke a number of actions, a list of which is displayed when you right-click within the Source pane. The following table describes the available actions.

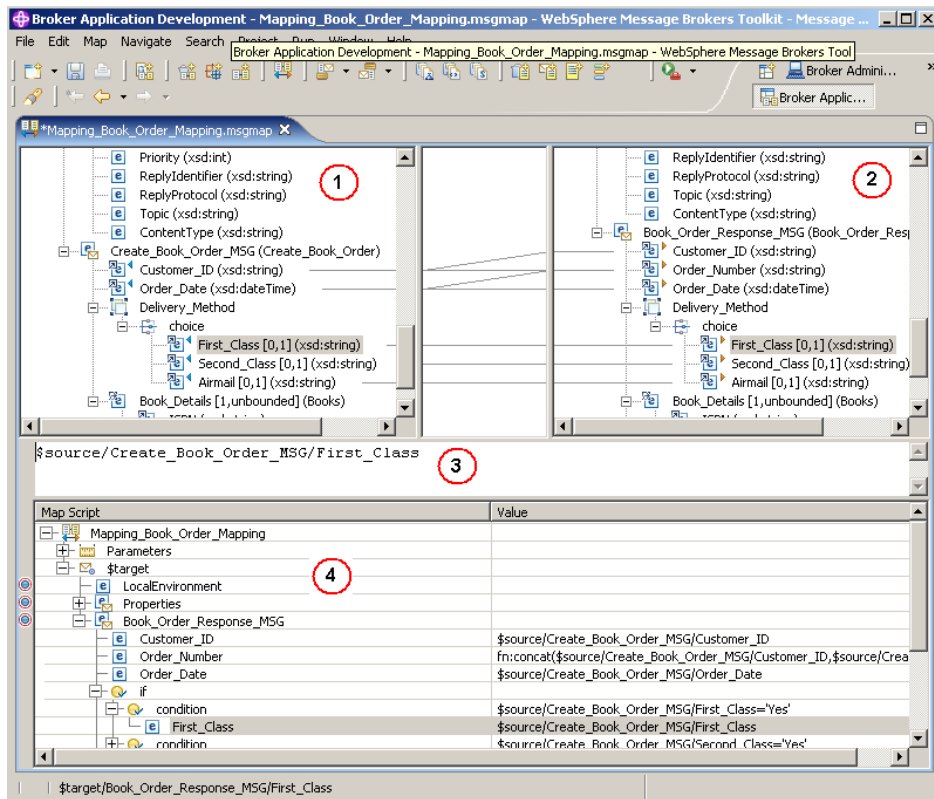
Action	Description	Related tasks
Undo	Undo previous action	
Redo	Redo previous action	
Revert	Discard	

Action	Description	Related tasks
Open Declaration (message)	<p>Display the element definition from the message set.</p> <p>For this action to be available, select any source message element except LocalEnvironment or Headers.</p>	
Open Declaration (database)	<p>Display the element definition from the database.</p> <p>For this action to be available, select any source database object.</p>	
Add Sources and Targets	<p>Add a message definition or a database table to a source.</p> <p>For this action to be available, select any source object.</p>	<p>“Adding a message to the source or target” on page 308, “Adding a database to the source” on page 309</p>
Go To	<p>For this action to be available, select any source object.</p>	
Delete (message)	<p>Remove a message and any existing maps from the source.</p> <p>For this action to be available, select the source message root (\$source).</p>	
Delete (database)	<p>Remove a database and any existing maps from the source.</p> <p>For this action to be available, select the source database root (\$db:select).</p>	
Map from Source	<p>Create a map between the focus source element and the focus target element.</p> <p>For this action to be available, select compatible source and target elements.</p>	<p>“Mapping a target element from source message elements” on page 310</p>
Accumulate	<p>If the source and target fields contain numeric data types, this action maps all occurrences of a repeating source field to a non-repeating target, resulting in the sum of all the source elements.</p> <p>For this action to be available, select the source and target element.</p>	<p>“Configuring a repeating source and a non-repeating target” on page 315</p>

Action	Description	Related tasks
Create New Submap	For this action to be available, select source and target elements that are either elements of complex types or wildcard elements.	
Call Existing Submap	Call an existing submap	
Call ESQL Routine	Call an ESQL routine	
Save	Save the .msgmap file	

Message Mapping editor Target pane

The following example shows the “Message Mapping editor” on page 748. The pane that is labelled as 2 in the example is the Target pane:



The following list describes the elements that are present in the Target pane:

- A target message is identified by \$target.
- A mapped entry is indicated by a yellow triangle alongside the element. In this example, Customer_ID, Order_Number, and Order_Date are mapped.
- Square brackets contain minimum and maximum occurrences of an element.
- An optional field is indicated by [0,1]. In this example, First_Class is optional.
- A repeating field is indicated by [minoccurs, maxoccurs].
- A choice field is indicated by a choice line; under the choice line are the possible choices. In this example, First_Class, Second_Class, and Airmail are choices of Delivery_Method.
- The type of each element is indicated in round brackets after the element name.

- If the message schema uses namespaces, the namespace prefix is shown before the element name, separated by a colon.

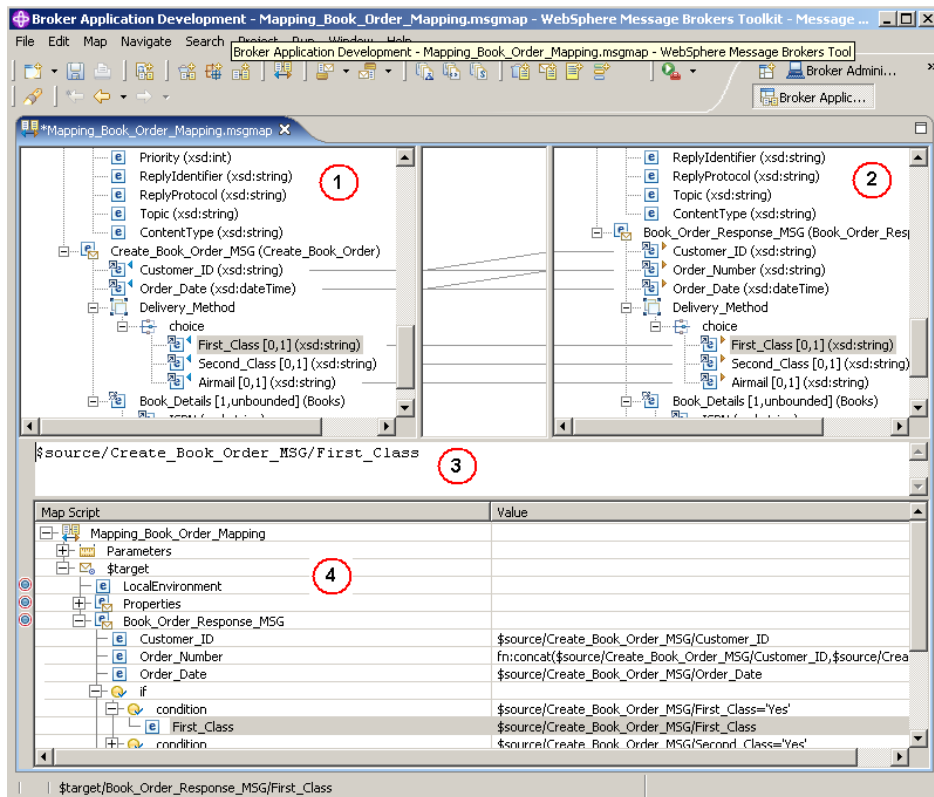
Use the Target pane to invoke a number of actions, a list of which is displayed when you right-click within the Target pane. The following table describes the available actions.

Action	Description	Related tasks
Undo	Undo previous action	
Redo	Redo previous action	
Revert	Discard	
Open Declaration (message)	Display the element definition from the message set. For this action to be available, select any target message element except LocalEnvironment or Headers.	
Add Sources and Targets	Add a message definition or a database table to a source. For this action to be available, select any target object.	“Adding a message to the source or target” on page 308, “Adding a database to the source” on page 309
Go To	For this action to be available, select any target object.	
Delete (message)	Remove a message and any existing maps from the source. For this action to be available, select the target message root (\$target).	
Map from Source	Create a map between the focus source element and the focus target element. For this action to be available, select compatible source and target elements.	“Mapping a target element from source message elements” on page 310
Enter Expression	For this action to be available, select any target object except \$target	“Setting the value of a target element to a constant” on page 312, “Setting the value of a target element using an expression or function” on page 313

Action	Description	Related tasks
Accumulate	If the source and target fields contain numeric data types, this action maps all occurrences of a repeating source field to a non-repeating target, resulting in the sum of all the source elements. For this action to be available, select the source and target element.	“Configuring a repeating source and a non-repeating target” on page 315
Create New Submap	For this action to be available, select source and target elements that are either elements of complex types or wildcard elements.	
Call Existing Submap	Call an existing submap	
Call ESQL Routine	Call an existing ESQL routine	
Save	Save the .msgmap file	

Message Mapping editor Edit pane

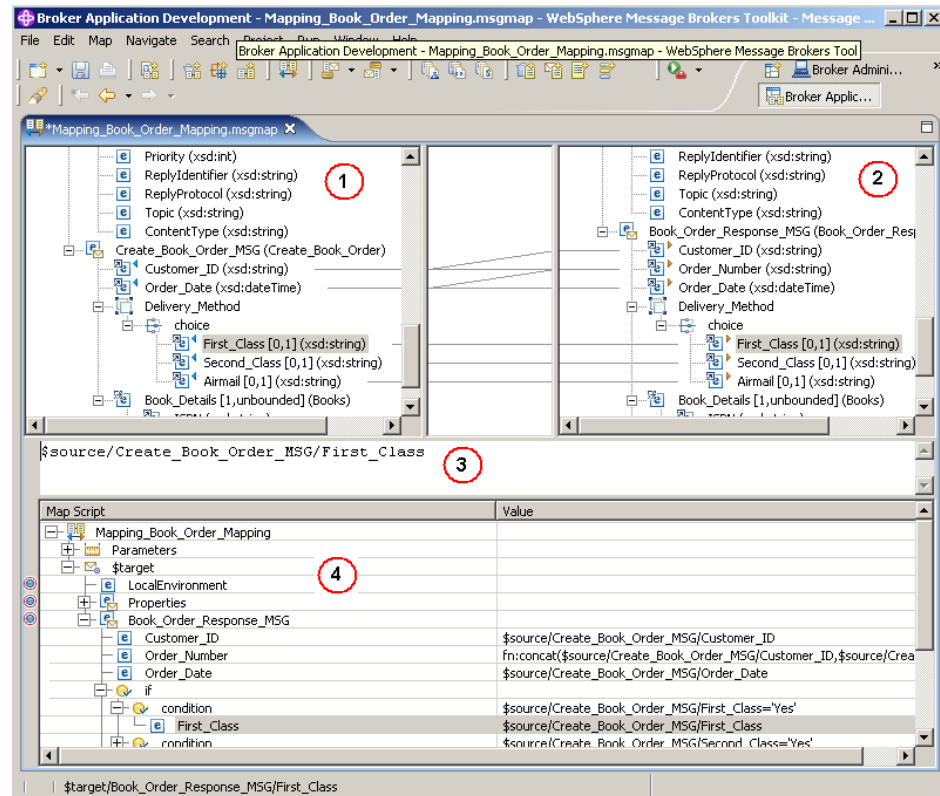
The following example shows the “Message Mapping editor” on page 748. The pane that is labeled as 3 in the example is the Edit pane:



When you have selected a source or target element, use the Edit pane to enter an expression. Right-click inside the Edit pane to invoke a list of available actions, most of which are standard Windows functions, such as cut, copy, and paste. Click **Edit** → **Content Assist** (or press Ctrl+Space) to access ESQL Content Assist, which provides a drop-down list of functions that are available in a Mapping node.

Message Mapping editor Spreadsheet pane

The following example shows the “Message Mapping editor” on page 748. The pane that is labelled as 4 in the example is the Spreadsheet pane:



Use the Spreadsheet pane to invoke a number of actions, a list of which is displayed when you right-click within the Spreadsheet pane. The following table describes the available actions.

Action	Description	Related tasks
Undo	Undo previous action	
Redo	Redo previous action	
Revert	Discard	
Add Sources and Targets	Add a message definition to a target. For this action to be available, select any target object.	“Adding a message to the source or target” on page 308, “Adding a database to the source” on page 309
Copy	Copy the selected item to the clipboard.	

Action	Description	Related tasks
Paste	Paste the item from the clipboard.	
Delete	Remove a row from the Spreadsheet.	
For	Define a repeating condition.	“Configuring a repeating source and a non-repeating target” on page 315 “Configuring a repeating source and a repeating target” on page 316
If	Placeholder for a new Condition block, to contain one or more Conditions.	“Configuring a repeating source and a non-repeating target” on page 315 “Configuring conditional mappings” on page 314
Condition	Define what must evaluate to ‘true’ to execute subsequent mappings.	“Configuring a repeating source and a non-repeating target” on page 315 “Configuring conditional mappings” on page 314
Else	Placeholder to execute subsequent mappings if previous Condition does not evaluate to ‘true’.	“Configuring conditional mappings” on page 314
Select Data Source	Define a database to be used in the mapping.	
Populate	Expand a structure so that each of its children have a row in the spreadsheet.	
Add Instance	Create a new row in the spreadsheet to set the value of a specific instance of a repeating field.	“Configuring a non-repeating source and a repeating target” on page 315
Add Group Instances	Create a number of new rows in the spreadsheet to set the values of specific instances of a repeating field.	“Configuring a non-repeating source and a repeating target” on page 315
Substitute Element	Select an element that: <ul style="list-style-type: none"> • belongs to the same substitution group • is derived from the same type 	
Save	Save the .msgmap file.	

Mapping node

The Mapping node has one or more mappings, which are stored in message map files (.msgmap). These files are configured using the “Message Mapping editor” on page 748.

A Mapping node must contain the following inputs and outputs:

- zero or one source (input) messages

- zero or more source (input) databases
- one or more target (output) messages

The source and target messages to be mapped must be defined in message definition files in a message set. The parser of the source message can be specified at run time (for example, in an MQRFH2 header), but the target message is built using the runtime parser that is specified in the message set.

If a message mapping is between elements of different types, you might need to include casts in your mapping definitions, depending on which runtime parser is specified in your message set.

The Mapping node uses a language to manipulate messages that are based on XPath.

To develop message mappings for a Mapping node, use the Message Mapping editor, which provides separate panes for working with sources, targets and expressions.

Mapping node syntax

In a Mapping node, the source message, if present, is identified in the “Message Mapping editor” on page 748 by \$source.

The message tree is represented in XPath format. For example, if you have an element called Body within a source message called Envelope, this is represented in the Mapping node as:

```
$source/soap11:Envelope/soap11:Body
```

Where *soap11* is a namespace prefix.

The first target message is identified by \$target; additional target messages are identified by \$target_1, \$target_2, etc.

The first source database is identified by \$db:select; additional source databases are identified by \$db:select_1, \$db:select_2, etc.

The database element is represented in the following format:

```
$db:select.DB.SCH.TAB.COL1
```

where:

DB is the database name
SCH is the database schema name
TAB is the table name
COL1 is the column name

You can also use the Mapping node to:

- make comparisons
- perform arithmetic
- create complex conditions

The comparison operators are:

= equals
 != not equals
 > greater than

- >= greater than or equals
- < less than
- <= less than or equals

The arithmetic operators are:

- + plus
- minus
- * multiply
- div divide

Conditional operators 'or' and 'and' are supported (these are case-sensitive).

The following objects can be mapped:

- Local Environment
 - MQDestination (single destination only)
 - HTTP Destination
 - RouterList
- Message headers (optional)
 - MQ Headers
 - HTTP Headers
 - JMSTransport
- Message elements
- Database columns

Mapping node functions

Various functions are available in a Mapping node. These are either provided within the Mapping node or user-defined. The functions that are provided with the Mapping node are of the following types:

- ESQL - prefixed esql:
- XPath - prefixed fn:
- Mapping - prefixed msgmap:
- Schema casts - prefixed xs:

Not all ESQL functions can be used in a Mapping node. For information about which functions are supported, and for a description of how to achieve equivalent processing for ESQL functions that are not supported, see the ESQL topics.

Mapping node casts

Source and target elements can be of different types in a Mapping node,

Depending on which runtime parsers are used, automatic casting cannot be done. In these cases, use one of the following cast functions:

- xs:boolean
- xs:date
- xs:dateTime
- xs:dayTimeDuration
- xs:decimal
- xs:duration
- xs:double
- xs:hexBinary

- xs:int
- xs:integer
- xs:string
- xs:long
- xs:time
- xs:yearMonthDuration

Migration of message mappings from Version 5.0

Use the `mqsimigratemfmaps` command to migrate message mappings. This command is part of the Message Brokers Toolkit, not the run time on Windows and Linux, and is not available in the command path by default. You can find the command under the Eclipse directory of the tooling installation. There are some restrictions on migrating message mappings.

The following table lists the mapping functions that are supported in Version 5.0 but not supported in Version 6.0, and shows the error messages that you might see. Mappings that contain these Version 5.0 functions cannot be migrated to Version 6.0, and must be re-created and redeployed using another node, such as a Java Compute node. Alternatively, try to migrate as much of the mapping as possible using the migration command, view the error report to see details of the functions that could not be migrated, and create a new node that will execute the non-migrated functions.

Supported in Version 5.0	Migration utility error message
Expressions that involve multiple instances of a repeating source element, for example: <code>src_msg.e[1] + src_msg.e[2] -> tgt_msg.e</code>	Error:102: Unexpected index '2' encountered for target mappable 'e'. The expected index is '1'. Migration currently provides no support for expressions involving more than one instance of the same repeating-element.
ESQL field references that contain the asterisk wildcard character "*". For example: <code>src_msg.e.*</code> or <code>src_msg.e.*[]</code>	Error:130: ESQL field-reference 'src_msg.e.*' cannot be migrated. Migration currently provides no support for field-references containing '*'.
Dynamic ESQL field references. For example: <code>src_msg.e.{ 'a' 'b' }</code>	Error:131: ESQL field-reference 'src_msg.e.{ 'a' 'b' }' cannot be migrated. Migration currently provides no support for dynamic field-references.
ESQL expressions that contain a reference to the temporary index-variable "#I". For example: <code>src_msg_e "#I" -> tgt_msg.e</code>	Error:128: ESQL expressions containing the variable '#I' anywhere other than the index of a repeating-element cannot be handled by the migration.
Expressions within an index of a repeating element. For example: <code>src_msg.e[src_msg.a]</code> or <code>src_msg.e["#I" +5]</code> or <code>src_msg.e[< 3]</code>	Error:116: ESQL field-reference 'src_msg.e[< 3]' cannot be migrated. Migration currently provides no support for indexes other than the variable '#I' and plain integer indexes.

Supported in Version 5.0	Migration utility error message
Aggregation functions MIN, MAX, and COUNT, used with the ESQL SELECT expression. For example: <pre>SELECT MAX("#T".FIRSTNAME) FROM Database.CUSTOMER AS "#T" WHERE "#T".CUSTOMERID = 7</pre>	Error:135: The ESQL expression 'SELECT MAX("#T".FIRSTNAME) FROM Database.CUSTOMER AS "#T" WHERE "#T".CUSTOMERID = 7' could not be migrated. The expression contains syntax which has no direct equivalent in the new map-script language.
ESQL's IN operator. For example: <pre>src_msg.e IN (1, 2, 3)</pre>	Error:135: The ESQL expression 'SELECT MAX("#T".FIRSTNAME) FROM Database.CUSTOMER AS "#T" WHERE "#T".CUSTOMERID = 7' could not be migrated.

Restrictions on migrating message mappings

There are certain scenarios where the migration of .mfmap files is not supported. This topic explains why migration is not automatic in these situations, and provides instructions for how to complete a successful migration.

The programming model for message maps is different between Version 5.0 (where the file format is .mfmap) and Version 6.0 (where the format is .msgmap). Version 5.0 message maps have a procedural programming model, which is essentially an alternative ESQL, where you describe all the steps that are required to perform a transformation. Version 6.0 uses a declarative programming model, where you describe the result of the transformation, and the tools determine how to achieve that result.

Most migration failures result from message maps that contain too much information about the steps that perform the transformation, and not enough information about the desired result. For these message maps, migration is enabled by changing the .mfmap file so that specific "how to" sections are separated into an ESQL function or procedure that can be called by the message map. The .mfmap file calls the ESQL instead of containing it as an expression. The **mqsimigratemfmaps** command then migrates the .mfmap file, but calls the ESQL instead of logging a migration error.

A limitation is that ESQL (the run time for .mfmap and .msgmap files) cannot define functions that return complex element (or REFERENCE) values. The following procedure explains how to work around this complex element target limitation; in many cases, it means that the map must be rewritten as ESQL. For more examples and information about calling ESQL from maps, see the WebSphere Message Brokers mapping sample at **Help** → **Samples Gallery** → **Technology samples** → **Message Brokers** → **Message Map**.

1. Determine whether you can define an ESQL function for the .mfmap file.
 - a. When the target value is a complex element, or in ESQL terms a REFERENCE, the individual mapping must be rewritten in the .msgmap file. Delete the mapping from the .mfmap file, and proceed to Step 4.
 - b. Use a function for all other cases: CHAR string, numbers, date and time. Proceed to Step 2.
2. Determine the source parameters and returns type for your function.
 - a. For each source path in the mapping, there must be one parameter in the function or procedure. For a function, all parameters are unchangeable. The type of the parameter must match the source data type.

- b. The function return type is the ESQL datatype identified above.
3. Update the .mfmap file to enable migration. Change the .mfmap file to invoke the function in the mapping, passing the source parameters to the function in the order in which they were listed in step 2a.
4. Re-run the **mqsimigratemfmaps** command to migrate the modified .mfmap file.
5. Repeat Steps 1 to 4 until there are no errors in the migration log.
6. Start the Version 6.0 Message Brokers Toolkit and open the migrated .msgmap file.
 - a. For ESQL that is migrated as functions, there should be no errors.
 - b. For complex element targets, rewrite the mapping using the Version 6.0 features.

The following examples illustrate migration of .mfmap files to .msgmap files.

- To migrate a multiple reference to a repeating source expression:

```
src_msg.e[1] + src_msg.e[2]
```

compute the result in an ESQL function like:

```
CREATE FUNCTION addOneAndTwo(IN src_msg)
BEGIN
  RETURN src_msg.e[1] + src_msg.e[2];
END;
```

In the .msgmap file, call the ESQL function addOneAndTwo using the parent element src_msg as a parameter.

- An expression that does not use element names:

```
src_msg.*
```

or

```
src_msg.*[]
```

can be processed using a function that takes the parent of the repeating field:

```
CREATE FUNCTION processAny(IN src_msg)
BEGIN
  DECLARE nodeRef REFERENCE TO src_msg.e.*;
  DECLARE result <dataType> <initialValue>;
  WHILE LASTMOVE nodeRef DO
    --expression goes here
    SET result = result;
  END WHILE;
  RETURN RESULT;
END;
```

In the .msgmap file, call the ESQL function using the parent element src_msg as a parameter.

- Expressions that dynamically compute element names:

```
src_msg.{ 'a' || 'b' }
```

can be processed by ESQL functions that process the parent of the repeating field:

```
CREATE FUNCTION processDynamicName(IN src_msg)
BEGIN
  RETURN src_msg.{ 'a' || 'b' };
END;
```

In the .msgmap file, call the ESQL function using the parent element src_msg as a parameter.

- Expressions that use the select MIN, MAX, and COUNT functions:

```
SELECT MAX("#T".FIRSTNAME)
FROM Database.CUSTOMER AS "#T"
WHERE "#T".CUSTOMERID = custId
```

can be processed by ESQL functions that process the parent of the repeating field:

```
CREATE FUNCTION processMAX(IN custId)
BEGIN
  RETURN
  SELECT MAX("#T".FIRSTNAME)
  FROM Database.CUSTOMER AS "#T"
  WHERE "#T".CUSTOMERID = custId
END;
```

In the .msgmap file, call the ESQL function using the element custId as a parameter.

- .mfmap files that use mfmap index variables in expressions:

```
e || "#I"
```

must be rewritten entirely in ESQL. By definition, there must be a complex repeating parent element, and this is not supported by ESQL functions.

- Expressions that use source expressions to compute values:

```
src_msg.e[src_msg.a]
```

must be rewritten using if rows, msgmap:occurrence() functions, and ESQL functions:

```
for src_msg.e
  if
    condition msgmap:occurrence(src_msg/e) = src_msg/a
```

- For expressions that use index expressions to compute values:

```
src_msg.e["#I" +5]
src_msg.e[< 3]
```

the entire .mfmap file must be rewritten in ESQL, because the .msgmap files do not yet support indexed access to repeating fields.

- .mfmap files that use ROW expressions to compute values:

```
src_msg.e IN (1, 2, 3)
```

must be rewritten in ESQL, because .msgmap files do not support ESQL ROW expressions.

Related tasks

“Developing ESQL” on page 145

XML constructs

A self-defining XML message carries the information about its content and structure within the message in the form of a document that adheres to the XML specification. Its definition is not held anywhere else. When the broker receives an XML message, it interprets the message using the generic XML parser, and created an internal message tree structure according to the XML definitions contained within that message.

A self-defining message is also known as a generic XML message. It does not have a recorded format.

The information provided with WebSphere Message Broker does not provide a full definition or description of XML terminology, concepts, and message constructs: it is a summary that highlights aspects that are important when you use XML messages with brokers and message flows.

For further information about XML, see the developerWorks Web site.

Example XML message

The name elements used in this description (for example, XmlDecl) are provided by WebSphere Message Broker, and are referred to as correlation names. They are available for symbolic use within the ESQL that defines the processing of message content performed by the nodes, such as a Filter node, within a message flow. They are not part of the XML specification.

A simple XML message might take the form:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

The corresponding syntax element tree (top level elements only) is shown below.



The WhiteSpace elements within the tree are there because of the line breaks in the original XML document, and have no business meaning. White space is used in XML for readability; if you process XML messages that contain line breaks (as shown above), blanks lines, or spaces between tags, these all appear as elements in the message tree.

WhiteSpace within an XML element (between start and end tags) has business meaning and is represented using the Content syntax element. See “XML WhiteSpace and DocTypeWhiteSpace” on page 774 for more information.

The correlation names for XML name elements (for example, Element and XmlDecl) equate to a constant value of the form 0x01000000. You can see these constants in the output created by the Trace node when a message, or a portion of the message, is traced.

The XML declaration

The beginning of an XML message can contain an XML declaration. An example of a declaration is shown below.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

The XML declaration includes the following correlation names:

- “XML encoding” on page 764
- “XML standalone” on page 764
- “XML version” on page 764

- “XMLDecl”

“XML declaration example” includes another example of an XML declaration and the tree structure it forms.

XML encoding

The encoding element is a value element and is always a child of the XmlDecl element. The value of the encoding element is a string that corresponds to the value of the encoding string in the declaration. In the example shown below, the encoding element has a value of UTF-8.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

You cannot specify WebSphere MQ encodings in this element.

XML standalone

The XML standalone element defines the existence of an externally-defined DTD. It is a value element and stores the data corresponding to the value of the standalone string in the declaration. It is always a child of the XmlDecl element. Valid values for the standalone element are yes and no. An example is shown below:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

A value of no indicates that this XML document is not standalone and depends on an externally-defined DTD. A value of yes indicates that the XML document is self-contained. However, the current release of WebSphere Message Broker does not resolve externally-defined DTDs, so the setting of standalone is irrelevant and is ignored.

XML version

The XML version element is a value element and stores the data corresponding to the version string in the declaration. It is always a child of the XmlDecl element. In the example below, the version element contains the string value 1.0:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

XMLDecl

XMLDecl is a name element that corresponds to the XML declaration itself. The XmlDecl element is a child of the XML parser and is written first to a bit stream. Although the XMLDecl element is a named element, its name has no relevance. An example is shown below:

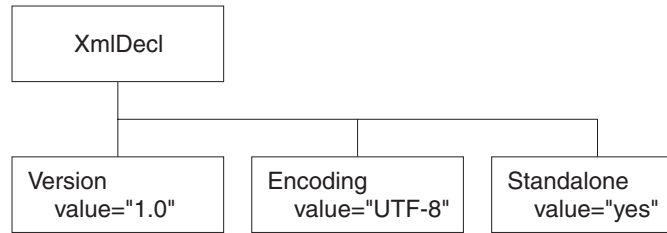
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

XML declaration example

The following example shows an XML declaration in an XML document.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

The following figure shows the tree structure that is created from the declaration:



The XML message body

Every XML message must have a body. The body comprises a top-level XML element that contains all the message data. The body contains complex XML markup, which translates to many syntax element types in the parsed tree. Each syntax element type is introduced here, with a series of example XML fragments.

The following common element types are discussed:

- “XML element” on page 767
- “XML attribute” on page 766
- “XML content” on page 767

“XML message body example: elements, attributes, and content” on page 768 provides an example of an XML message body and the tree structure that is created from it using the syntax elements types listed above.

More complex XML messages might use some of the following syntax element types:

- “XML CDataSection” on page 766
- “XML EntityReferenceStart and EntityReferenceEnd” on page 768
- “XML comment” on page 767
- “XML ProcessingInstruction” on page 768
- “XML AsIsElementContent”
- “XML BitStream” on page 766

XML AsIsElementContent

The AsIsElementContent syntax element is a special value element. It is used to precisely control the XML generated in an output message without the safeguards of the Element, Attribute, and Content syntax elements. If you use AsIsElementContent, you must ensure that the output message is well-formed XML.

You might choose to use this syntax element if, for example, you want to suppress the normal behavior in which occurrences of ampersand (&), less than (<), greater than (>), double quote ("), and apostrophe (') are replaced by the predefined XML entities &, <, >, ", and '.

The following example illustrates the use of AsIsElementContent. The statement:

```
Set OutputRoot.XML.(XML.Element)Message.(XML.Content) = '<rawMarkup>';
```

generates the following XML in an output message:

```
<Message>&lt;rawMarkup&gt;</Message>
```

However, the statement

```
Set OutputRoot.XML.(XML.Element)Message.(XML.AsIsElementContent) = '<rawMarkup>';
```

generates the following output message:

```
<Message><rawMarkup></Message>
```

This shows that the value of an `AsisElementContent` syntax element is not modified before it is written to the output message.

XML attribute

This syntax element is the default name-value element supported by the XML parser. Use it to represent the attributes associated with its parent element. The name and value of the syntax element correspond to the name and value of the attribute being represented. Attribute elements have no children, and must always be children of an element.

When attributes are written to a message, occurrences of ampersand (&), less than (<), greater than (>), double quote ("), and apostrophe (') within the attribute value are replaced by the predefined XML entities `&`, `<`, `>`, `"`, and `'`.

The `attr` element is also supported for backward compatibility.

XML BitStream

This syntax element is a name-value element. When writing an XML message, the value of the `BitStream` element is written directly into the message, and the name is not important. The `BitStream` element might be the only element in the message tree.

The value of the element must be of type `BLOB`; any other data type generates an error while writing the element. Ensure that the content of the element is appropriate for use in the output message.

Use of the `BitStream` element is similar to the `AsisElementContent` element, except that the `AsisElementContent` type converts its value into a string, whereas the `BitStream` element uses its `BLOB` value directly. This is a specialized element designed to aid processing of very large messages.

The following ESQL excerpts demonstrate a typical use for this element. First, declare the element:

```
DECLARE StatementBitStream BLOB
```

Initialize the contents of `StatementBitStream` from an appropriate source, such as an input message. If the source field is not of type `BLOB`, use the `CAST` statement to convert the contents to `BLOB`. Then create the new field in the output message, for example:

```
CREATE LASTCHILD OF resultCursor  
  Type XML.BitStream  
  NAME 'StatementBitStream'  
  VALUE StatementBitstream;
```

XML CDataSection

`CData` sections in the XML message are represented by the `CDataSection` value element. The content of the `CDataSection` element is the value of the `CDataSection` element without the `<![CDATA[` that marks its beginning and the `]]>` that marks its end.

For example, the following `Cdata` section:

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

is represented by a CDATASection element with a string value of:

```
"<greeting>Hello, world!</greeting>"
```

Unlike Content, occurrences of <, >, &, ", and ' are not translated to their escape sequences when the CDATASection is written out to a serialized message (bit stream).

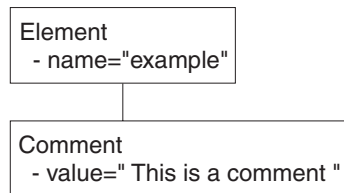
XML comment

An XML comment encountered outside the document type declaration is represented by the Comment value syntax element. The value of the element is the comment text from the XML message.

If the value of the element contains the character sequence -->, the sequence is replaced with the text -->. This ensures that the contents of the comment cannot prematurely terminate the comment. Occurrences of <, >, &, ", and ' are not translated to their escape sequences.

Examples of the XML comment in an XML document and in tree structure form are shown below:

```
<example><!-- This is a comment --></example>
```



XML content

This syntax element is the default value element supported by the XML parser. Use content to represent character data (including whitespace) that is part of the element content. There might be many content elements as children of a single element, in which case they are separated by other syntax element types such as nested elements or attributes.

When content is written to a message, occurrences of ampersand (&), less than (<), greater than (>), double quote ("), and apostrophe (') are replaced by the predefined XML entities &, <, >, ", and '.

The pCDATA element is also supported for backward compatibility.

XML element

This syntax element is the default name element supported by the XML parser, and is one of the most common elements. The name of the syntax element corresponds to the name of the XML element in the message. This element can have many children, including attributes, elements, and content.

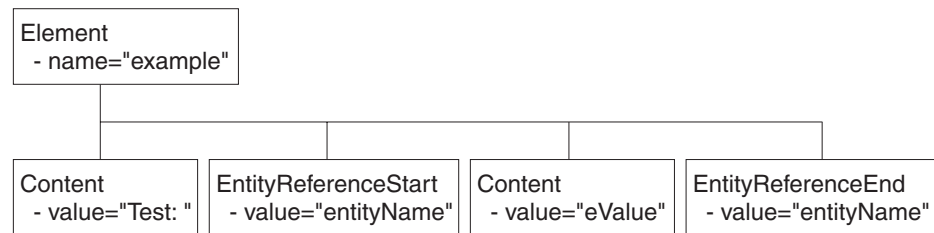
The tag element is also supported for backward compatibility.

XML EntityReferenceStart and EntityReferenceEnd

When an entity reference is encountered in the XML message, both the expanded form and the original entity name are stored in the syntax element tree. The name of the entity is stored as the value of the EntityReferenceStart and EntityReferenceEnd syntax elements, and any syntax elements between contain the entity expansion.

Examples of the XML entity references in an XML document and in tree structure form are shown below:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE example [ <!ENTITY entityName "eValue"> ]>
<example>Test: &entityName;</example>
```

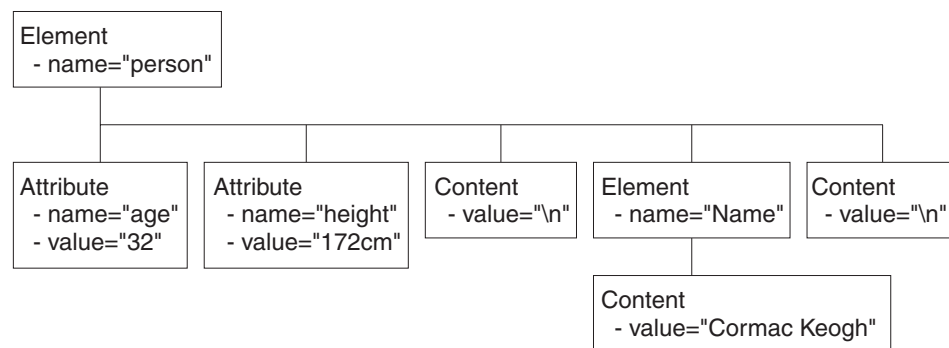


The XML declaration and the document type declaration are not shown here. Refer to “The XML declaration” on page 763 and “XML document type declaration” on page 769 for details of those sections of the syntax element tree.

XML message body example: elements, attributes, and content

Examples of an XML message body in an XML document and in tree structure form are shown below. The XML document contains elements, attributes, and content, and these items are shown in the tree structure.

```
<Person age="32" height="172cm">
<Name>Cormac Keogh</Name>
</Person>
```



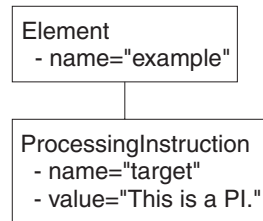
XML ProcessingInstruction

An XML processing instruction encountered outside the document type declaration is represented by the ProcessingInstruction syntax element. This is a name-value element; the name of the syntax element is the processing instruction target name, and the value of the syntax element is the character data of the processing instruction. The value of the syntax element must not be empty. The name cannot be XML in either uppercase or lowercase.

If the value of the element contains the character sequence `>`, the sequence is replaced with the text `>`. This ensures that the content of the processing instruction cannot prematurely terminate the processing instruction. Occurrences of `<`, `>`, `&`, `"`, and `'` are not translated to their escape sequences.

Examples of the XML ProcessingInstruction in an XML document and in tree structure form are shown below:

```
<example><?target This is a PI.?></example>
```



XML document type declaration

The document type declaration (DTD) of an XML message is represented by a syntax element of type `DocTypeDecl` and its children and descendants. These comprise the `DOCTYPE` construct.

Only internal (inline) DTD subsets are represented in the syntax element tree. An inline DTD is a DTD that is declared within the XML document itself. It can be a complete DTD definition, or can extend the definition in an external DTD.

External DTD subsets (identified by the `SystemID` or `PublicId` elements described below) can be referenced in the message, but those referenced are not resolved by the broker.

Correlation names are defined by WebSphere Message Broker:

- `DocTypeDecl`
- `NotationDecl`
- `Entities`
- `ElementDef`
- `AttributeList`
- `AttributeDef`
- `DocTypePI`
- `WhiteSpace` and `DocTypeWhiteSpace`
- `DocTypeComment`

DTD example is an example of an XML DTD.

XML DocTypeDecl

`DocTypeDecl` is a named element and is a child of the XML parser. `DocTypeDecl` is written to the bit stream before the element that represents the body of the document during serialization. The following attributes can be specified within this element:

- `IntSubset`
- `PublicId`
- `SystemId`

The example below is included in DTD example:

```
<!DOCTYPE test PUBLIC "-//this/is/a/URI/test" "test.dtd" [  
...  
...  

```

XML IntSubset:

IntSubset is a named element that groups all those elements that represent the DTD constructs contained in the internal subset of the message. Although the IntSubset element is a named element, its name is not relevant.

XML PublicId:

PublicId is an element that represents a public identifier in an XML message. It can be part of a DocTypeDecl, NotationDecl, or UnparsedEntityDecl element. The value of the PublicId element is typically a URL. A public identifier of the form PUBLIC "-//this/is/a/URI/test" has a string value of //this/is/a/URI/test.

XML SystemId:

SystemId is a value element that represents a system identifier in an XML message. It can be part of a DocTypeDecl, NotationDecl, or UnparsedEntityDecl element. The value of the SystemId is a URI, and is typically a URL or the name of a file on the current system. A system identifier of the form SYSTEM "Note.dtd" has a string value of Note.dtd.

XML NotationDecl

The NotationDecl element represents a notation declaration in an XML message. NotationDecl is a name element whose name corresponds to the name given with the notation declaration. It must have a SystemId as a child and it can optionally have a child element of type PublicId. For example:

```
<!NOTATION gif SYSTEM "image.gif">
```

The name of the NotationDecl is gif.

XML entities

Entities in the DTD are represented by one of six named element types described below:

- EntityDecl
- EntityDeclValue
- ExternalParameterEntityDecl
- ExternalEntityDecl
- ParameterEntityDecl
- UnparsedEntityDecl

XML EntityDecl:

The EntityDecl element represents a general entity and is declared in the internal subset of the DTD. It is a named element and has a single child element, which is of type EntityDeclValue.

An entity declaration of the form:

```
<!ENTITY bookTitle "User Guide">
```

has an EntityDecl element of name bookTitle and a child element of type EntityDeclValue with a string value of User Guide.

XML EntityDeclValue:

The EntityDeclValue element represents the value of an EntityDecl or ParameterEntityDecl defined internally in the DOCTYPE construct. It is always a child of an element of one of those types, and is a value element. For the following entity:

```
<!ENTITY bookTitle "User Guide">
```

the EntityDeclValue element has the string value User Guide.

XML ExternalParameterEntityDecl:

The ExternalParameterEntityDecl element represents a parameter entity definition where the entity definition is contained externally to the current message. It is a named element and has a child of type SystemId. It can also have a child of type PublicId. The name of the entity does not include the percent sign %. In XML an external parameter entity declaration takes the form:

```
<!ENTITY % bookDef SYSTEM "BOOKDEF.DTD">
```

This represents an ExternalParameterEntityDecl element of name bookDef with a single child of type SystemId with a string value of BOOKDEF.DTD.

XML ExternalEntityDecl:

The ExternalEntityDecl element represents a general entity where the entity definition is contained externally to the current message. It is a named element and has a child of type SystemId. It can also have a child of type PublicId.

An external entity declaration of the form:

```
<!ENTITY bookAppendix SYSTEM "appendix.txt">
```

has an EntityDecl element of name bookAppendix and a child element of type SystemId with a string value of appendix.txt.

XML ParameterEntityDecl:

The ParameterEntityDecl represents a parameter entity definition in the internal subset of the DTD. It is a named element and has a single child element that is of type EntityDeclValue. For parameter entities, the name of the entity does not include the percent sign %. In XML a parameter entity declaration takes the form:

```
<!ENTITY % inline "#PCDATA | emphasis | link">
```

XML UnparsedEntityDecl:

An unparsed entity is an external entity whose external reference is not parsed by an XML processor. This means that you can include data in an XML document that is not well-formed XML, such as a graphic file. The UnparsedEntityDecl is named element and a child of type SystemId that identifies the URI for the entity (a URL or a local file location). UnparsedEntityDecl can optionally have a child of type PublicId.

UnparsedEntityDecl can also have a child of type NotationReference, a value element that represents a reference to a notation declaration elsewhere in the XML document. It defines the type of data of the unparsed entity.

An unparsed entity declaration takes the form:

```
<!ENTITY pic SYSTEM "scheme.gif" NDATA gif>
```

In this example, the SystemId has a string value of scheme.gif. The value of NotationReference is gif. It refers to a NOTATION defined within the XML document:

```
<!NOTATION gif SYSTEM "image/gif">
```

The next entity is included in the DTD example:

```
<!ENTITY unpsd PUBLIC "-//this/is/a/URI/me.gif" "me.gif" NDATA TeX>
```

This shows the optional PublicId element, which has the string value of //this/is/a/URI/me.gif.

XML ElementDef

The ElementDef element represents the <!ELEMENT construct in a DTD. It is a child of the DOCTYPE element. The name of the element that is defined corresponds to the name of the syntax element. The value corresponds to the element definition.

This example is included in the DTD example:

```
<!ELEMENT sube12 (#PCDATA)>
```

The name of the element is sube12 and the value is (#PCDATA).

XML AttributeList

The AttributeList name element represents the <!ATTLIST construct in a DTD. The name of the AttributeList element corresponds to the name of the element for which the list of attributes is being defined. Its content represents one or more AttributeDef elements.

This example is included in the DTD example:

```
<!ATTLIST e15 e15satt CDATA #IMPLIED>
```

This example shows an AttributeList that defines one AttributeDef, and its content is explained in AttributeDef.

XML AttributeDef

The AttributeDef name element describes the definition of an attribute within an <!ATTLIST construct. It is always a child of the AttributeList element. The name of the syntax element is the name of the attribute being defined. It can have three children:

- AttributeDefValue
- AttributeDefType
- AttributeDefDefaultType

This example is included in the DTD example:

```
<!ATTLIST e15 e15satt CDATA #IMPLIED>
```

The name of the AttributeDef is e15satt and it is a child of AttributeList e15. The name of the AttributeDefType is CDATA, and the value of the AttributeDefDefaultType is IMPLIED.

XML AttributeDefValue:

For attributes of type CDATA (see “XML AttributeDefType”), or defined by an enumerated list, the AttributeDefValue gives the default value of the attribute.

For an example of AttributeDefValue, see DTD example.

XML AttributeDefType:

The AttributeDefType syntax element is a name-value element whose name corresponds to the attribute type found in the attribute definition. Possible values for the name are:

- CDATA
- ID
- IDREF
- IDREFS
- ENTITY
- ENTITIES
- NMTOKEN
- NMTOKENS
- NOTATION

If there is an enumeration present for the attribute definition, the entire enumeration string is held as a string in the value member of the name-value syntax element. In this case, the name member of the name-value syntax element is empty. The value string starts with an open parenthesis (and ends with a close parenthesis). Each entry in the enumeration string is separated by a vertical bar | character. If the Attribute value is not defined by an enumerated list, the value member of the syntax element is empty.

An example is included in AttributeDef.

XML AttributeDefDefaultType:

The AttributeDefDefaultType syntax element is a value element that represents the attribute default as defined under the attribute definition. The value can be one of the following strings:

- #REQUIRED
- #IMPLIED
- #FIXED

An example is included in AttributeDef.

XML DocTypeComment

Comments in the XML DTD are represented by the DocTypeComment element. It is a value element for which the value string contains the comment text. This element follows the same processing rules as the Comment element. See “XML comment” on page 767.

XML DocTypePI

The DocTypePI element represents a processing instruction found within the DTD. The ProcessingInstruction element represents a processing instruction found in the XML message body.

This element is a name-value element. The name of the element is used to store the processing instruction target name, and the value contains the character data of the processing instruction. The value of the element can be empty. The name cannot be the string XML in either uppercase or lowercase form. This element follows the same processing rules as the ProcessingInstruction element. See “XML ProcessingInstruction” on page 768.

XML WhiteSpace and DocTypeWhiteSpace

The WhiteSpace element represents any white space characters outside the message body and DTD that are not represented by any other element. For example, white space within the body of the message (within elements) is reported as element content using the Content element type, but white space characters between the XML declaration and the beginning of the message body are represented by the WhiteSpace element.

```
<?xml version="1.0"?>      <BODY>...</BODY>
```

The characters between "1.0"?> and <BODY> are represented by the WhiteSpace element.

White space is used in XML for readability and has no business meaning. Input XML messages can include line breaks, blanks lines, and spaces between tags (all shown below). If you process XML messages that contain any of these spaces, they are represented as elements in the message tree. Therefore they appear when you view the message in the debugger, and in any trace output.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....
<s2>abc</s2>   <s2>def</s2>

<s3>123</s3>
</s1>
```

If you do not want white space elements in your message trees, you must present the input message as a single line, or use the XMLNSC compact parser in its default mode

The DocTypeWhiteSpace element represents white space that is found inside the DTD that is not represented by any other element. White space characters found within a DocType between two definitions are represented by the DocTypeWhiteSpace element.

```
<!ENTITY % bookDef SYSTEM "BOOKDEF.DTD"> <!ENTITY bookTitle "User Guide">
```

The characters between DTD"> and <!ENTITY are represented by the DocTypeWhiteSpace element.

XML DTD example

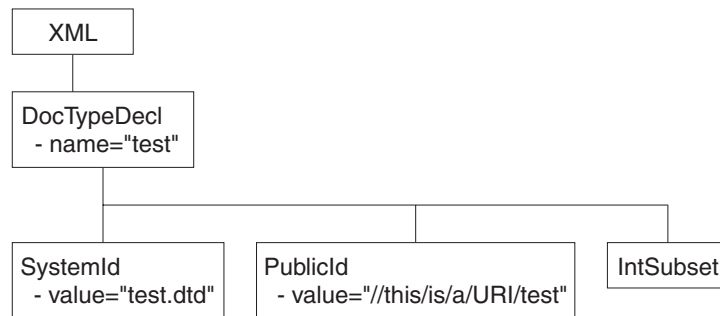
This example shows an XML DTD in an XML document and the tree structure form of that document:

```

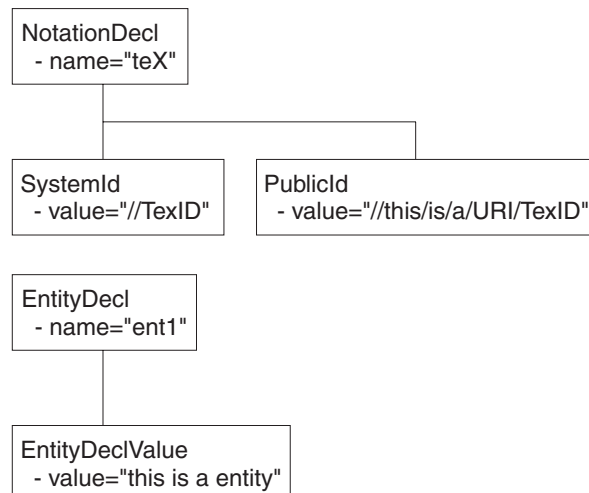
<!DOCTYPE test PUBLIC "-//this/is/a/URI/test" "test.dtd" [
<!NOTATION TeX PUBLIC "-//this/is/a/URI/TeXID" "//TeXID">
<!ENTITY ent1 "this is an entity">
<!ENTITY % ent2 "#PCDATA | sube12">
<!ENTITY % extent1 PUBLIC "-//this/is/a/URI/extent1" "more.txt">
<!ENTITY extent2 PUBLIC "-//this/is/a/URI/extent2" "more.txt">
<!ENTITY unpsd PUBLIC "-//this/is/a/URI/me.gif" "me.gif" NDATA TeX>
<?test Do this?>
<!--this is a comment-->
<ELEMENT sube12 (#PCDATA)>
<!ELEMENT sube11 (sube12 | e14)+>
<!ELEMENT e11 (#PCDATA)>
<!ELEMENT e12 (#PCDATA | sube12)*>
<!ELEMENT e13 (#PCDATA | sube12)*>
<!ELEMENT e14 (#PCDATA)>
<!ELEMENT e15 (#PCDATA | sube11)*>
<!ELEMENT e16 (#PCDATA)>
<!ATTLIST sube11
  size (big | small) "big"
  shape (round | square) #REQUIRED>
<!ATTLIST e15
  e15satt CDATA #IMPLIED>
]>

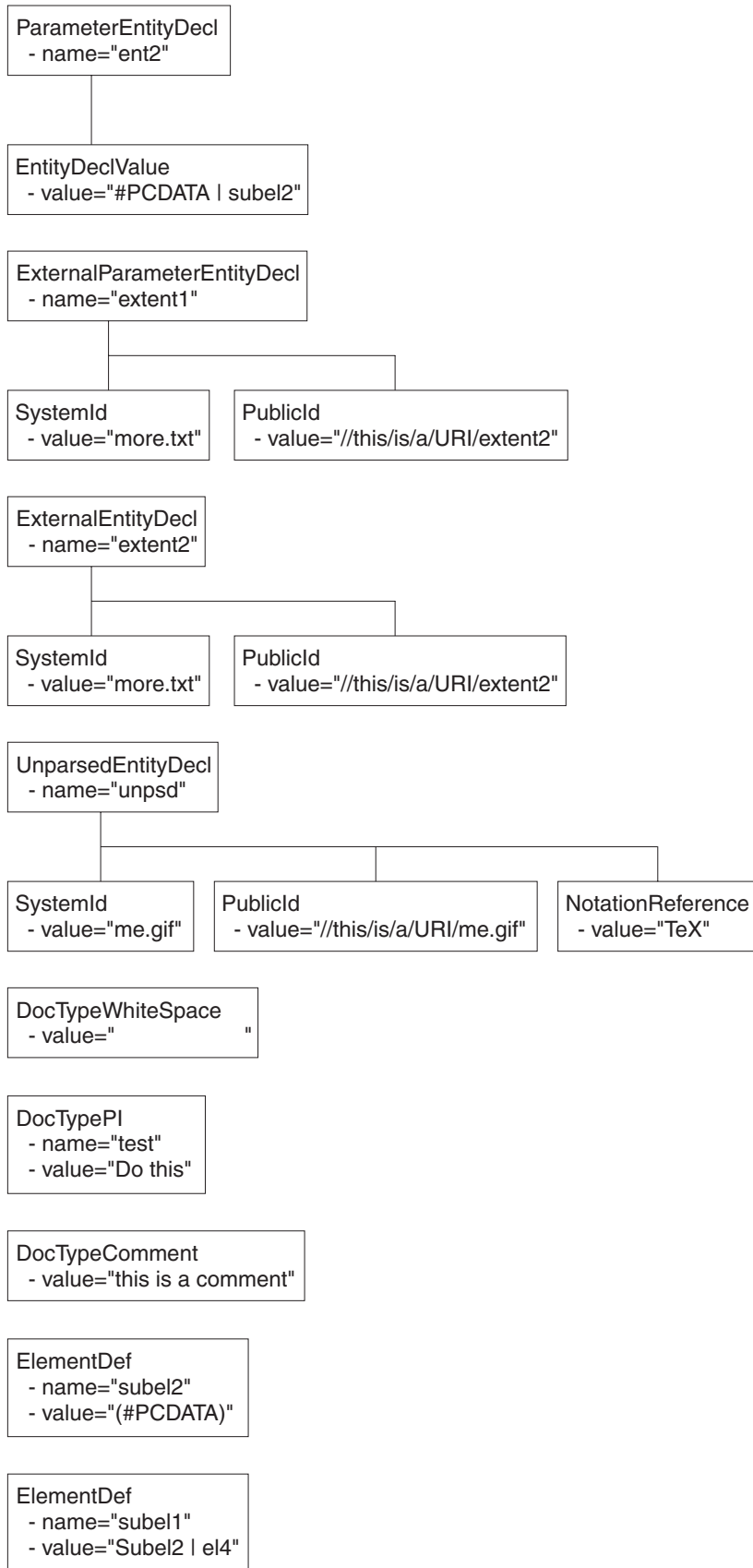
```

When a message is parsed by the generic XML parser, the relevant part of the message tree looks like this (assuming that there are no carriage returns or white space between tags):



The IntSubset structure contains the following structures at the next level of nesting: the tree structure for each of these is shown in the tree structures below.





ElementDef
- name="el1"
- value="(PCDATA)"

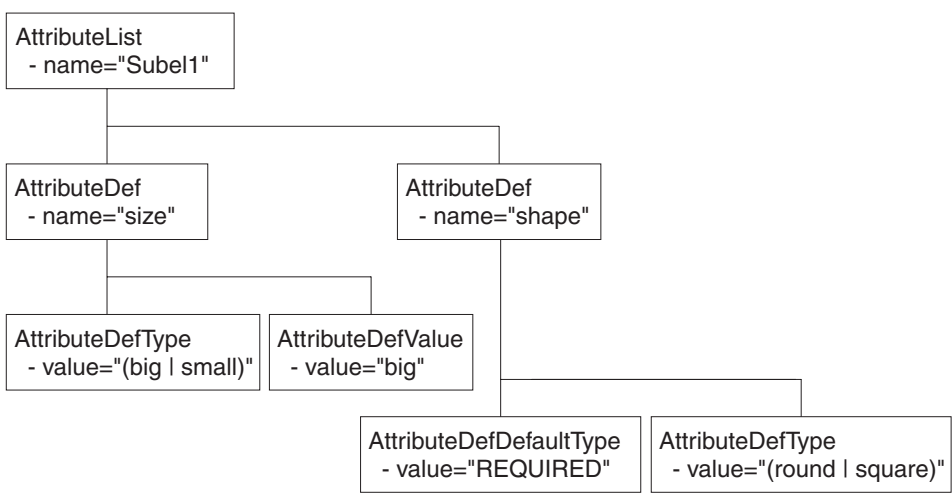
ElementDef
- name="el2"
- value="(PCDATA | Subel2)*"

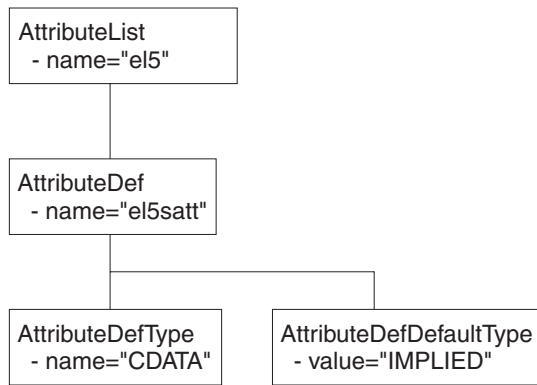
ElementDef
- name="el3"
- value="(PCDATA | Subel2)*"

ElementDef
- name="el4"
- value="(PCDATA)"

ElementDef
- name="el5"
- value="(PCDATA | Subel1)*"

ElementDef
- name="el6"
- value="(PCDATA)"





Data sources on z/OS

The Data Source name in the Compute and Database nodes identifies the location of the table referred to in the respective node's ESQL. Data sources on z/OS correspond to DB2 subsystems rather than DB2 databases. The DB2 owning region for a particular database table is identified using a combination of the DSNAOINI file and DB2 subsystem configuration.

The MVSDEFAULTSSID parameter in the DSNAOINI file identifies the local DB2 subsystem to which the broker is connected. This subsystem is used to locate the data source which is either a local or remote DB2. The mapping between a particular data source and DB2 subsystem is shown in the DSNTIPR installation panel of the default DB2 subsystem and SYSIBM.LOCATIONS table.

When you access remote DB2 subsystems, ensure that the DBRMs for ODBC are bound at the remote subsystem. For more information, refer to the 'Programming for ODBC' topics in the DB2 Information Management Software Information Center for z/OS Solutions .

If you need to access databases that are not on DB2 on z/OS, you can use DB2's Distributed Data Facility (DDF) and Distributed Relational Architecture (DRDA) to incorporate a remote unit of work within a message flow.

ESQL reference

SQL is the industry standard language for accessing and updating database data and ESQL is a language derived from SQL Version 3, particularly suited to manipulating both database and message data.

This section covers the following topics:

“Syntax diagrams: available types” on page 780

This describes the formats that are available for viewing ESQL syntax diagrams.

“ESQL data types in message flows” on page 780

This describes the valid data types for ESQL.

“ESQL field references” on page 792

This topic describes the syntax of field references.

“Special characters, case sensitivity, and comments in ESQL” on page 986

This describes the special characters you use when writing ESQL statements.

“ESQL operators” on page 798

This describes the operators that are available.

“ESQL reserved keywords” on page 988

This lists the reserved keywords which you cannot use for variable names.

“ESQL non-reserved keywords” on page 988

This lists the keywords that are not reserved, as well as those reserved for future releases, which you can use if you choose.

“ESQL functions: reference material, organized by function type” on page 889

This topic lists the functions available in ESQL, and what they do.

“ESQL statements” on page 804

This topic lists the different statement types available in ESQL, and what they do.

“Calling ESQL functions” on page 892

This topic describes all the ESQL functions in detail.

“ESQL variables” on page 791

This topic describes the types of ESQL variable and their lifetimes.

“Broker properties accessible from ESQL and Java” on page 983

This topic lists the broker attributes that can be accessed from ESQL code.

An XML format message that is used in many of the ESQL examples in these topics is shown in “Example message” on page 991.

For information about how you can use ESQL statements and functions to configure Compute, Database, and Filter nodes, see “Writing ESQL” on page 168.

Syntax diagrams: available types

The syntax for commands and ESQL statements and functions is presented in the form of a diagram. The diagram tells you what you can do with the command, statement, or function and indicates relationships between different options and, sometimes, different values of an option. There are two types of syntax diagrams: railroad diagrams and dotted decimal diagrams. Railroad diagrams are a visual format suitable for sighted users. Dotted decimal diagrams are text-based diagrams that are more helpful for blind or partially-sighted users.

To select which type of syntax diagram you use, click the appropriate button above the syntax diagram in the topic that you are viewing.

The following topics describe how to interpret each type of diagram:

- How to read railroad diagrams
- How to read dotted decimal diagrams

ESQL data types in message flows

All data that is referred to in message flows must be one of the defined types:

- “ESQL BOOLEAN data type”
- “ESQL datetime data types”
- “ESQL NULL data type” on page 786
- “ESQL numeric data types” on page 786
- “ESQL REFERENCE data type” on page 789
- “ESQL ROW data type” on page 781
- “ESQL string data types” on page 789

ESQL BOOLEAN data type

The BOOLEAN data type holds a boolean value which can have the values:

- TRUE
- FALSE
- UNKNOWN

Boolean literals consist of the keywords TRUE, FALSE, and UNKNOWN. The literals can appear in uppercase or lowercase. For further information about UNKNOWN, see the “IF statement” on page 863.

ESQL datetime data types

ESQL supports several data types that handle datetime values. The following data types are collectively known as **datetime** data types:

- “ESQL DATE data type” on page 781
- “ESQL TIME data type” on page 782
- “ESQL GMTTIME data type” on page 782
- “ESQL TIMESTAMP data type” on page 782
- “ESQL GMTTIMESTAMP data type” on page 782
- “ESQL INTERVAL data type” on page 783

For information about datetime functions see “ESQL datetime functions” on page 897.

ESQL DATE data type

The DATE data type holds a Gregorian calendar date (year, month, and day). The format of a DATE literal is the word DATE followed by a space, followed by a date in single quotation marks in the form 'yyyy-mm-dd'. For example:

```
DECLARE MyDate DATE;  
SET MyDate = DATE '2000-02-29';
```

Do not omit leading zeroes from the year, month, and day.

ESQL ROW data type

The ROW data type holds a **tree structure**. A row in a database is a particular type of tree structure, but the ROW data type is not restricted to holding data from database rows.

In a database, a row is a fixed, ordered, set of scalar values.

Note: A *scalar* is a single entity value or a string.

A database table is an unordered set of rows and is thus a two dimensional "array" of scalar values, in which one dimension is fixed and the other is variable. In ESQL, a row is an open-ended, ordered, set of named values in which each value can be scalar or another row. That is, a row is an open-ended tree structure with no restrictions on dimensions or regularity. Consider the following diagram:

```
Root  
  Row  
    PartNumber = 1  
    Description = 'Chocolate bar'  
    Price      = 0.30  
  Row  
    PartNumber = 2  
    Description = 'Biscuit'  
    Price      = 0.35  
  Row  
    PartNumber = 3  
    Description = 'Fruit'  
    Price      = 0.42
```

In the example, *Root* contains three elements all named "Row". Each of these in turn contains three elements with different names and values. This diagram equally describes an instance of an ESQL row data type (that is, a tree structure) or the contents of a database table.

ROW and LIST

The ROW data type is a normal data type. You can use the DECLARE statement to create ROW variables in the same way as you create INTEGER or CHARACTER variables. There is also a more general concept of a ROW data type. In the previous example, *Root* is the root element of a ROW variable. Each of the elements called "Row", while not the root element of ROW variables, are the root elements of sub-structures. Many ESQL operations (and particularly the SELECT function) work with the general concept of ROW and will operate equally on whole trees or parts of them.

There is also a general concept of a LIST data type. The set of elements called "Row" can be regarded as a list. Some ESQL operations (particularly SELECT) work with the general concept of list.

InputRoot, *OutputRoot* (and so on) are examples of ROW variables that are automatically declared and connected into the broker's structure, ready for use.

ESQL TIME data type

The TIME data type holds a time of day in hours, minutes, seconds, and fractions of a second. The format of a TIME literal is the word TIME followed by a space, followed by a time in single quotation marks in the form 'hh:mm:ss.ffffff'. For example:

```
DECLARE MyTime TIME;  
SET MyTime = TIME '11:49:23.656';
```

Each of the hour, minute, and second fields in a TIME literal must always be two digits; the optional fractional seconds field can be up to 6 digits in length.

The PutTime reported by WebSphere MQ on z/OS and other times or timestamps can be inconsistent if the CVT field is not set correctly. For details about when this problem can occur, and how to solve it, see The PutTime that is reported by WebSphere MQ on z/OS, and other times or timestamps are inconsistent.

ESQL GMTTIME data type

The GMTTIME data type is similar to the TIME data type, except that its values are interpreted as values in Greenwich Mean Time. GMTTIME literals are defined in a similar way to TIME values. For example:

```
DECLARE MyGetGmttime GMTTIME;  
SET MyGetGmttime = GMTTIME '12:00:00';
```

The PutTime reported by WebSphere MQ on z/OS and other times or timestamps can be inconsistent if the CVT field is not set correctly. For details about when this problem can occur, and how to solve it, see The PutTime that is reported by WebSphere MQ on z/OS, and other times or timestamps are inconsistent.

ESQL TIMESTAMP data type

The TIMESTAMP data type holds a DATE and a TIME in years, months, days, hours, minutes, seconds, and fractions of a second. The format of a TIMESTAMP literal is the word TIMESTAMP followed by a space, followed by a timestamp in single quotation marks in the form 'yyyy-mm-dd hh:mm:ss.ffffff'. For example:

```
DECLARE MyTimeStamp TIMESTAMP;  
SET MyTimeStamp = TIMESTAMP '1999-12-31 23:59:59';
```

The year field must always be four digits in length. The month, day, hour, and minute fields must always be two digits. (Do not omit leading zeroes.) The optional fractional seconds field can be 0 - 6 digits long.

The PutTime reported by WebSphere MQ on z/OS and other times or timestamps can be inconsistent if the CVT field is not set correctly. For details about when this problem can occur, and how to solve it, see The PutTime that is reported by WebSphere MQ on z/OS, and other times or timestamps are inconsistent.

ESQL GMTTIMESTAMP data type

The GMTTIMESTAMP data type is similar to the TIMESTAMP data type, except that the values are interpreted as values in Greenwich Mean Time. GMTTIMESTAMP values are defined in a similar way to TIMESTAMP values, for example:

```

DECLARE MyGetGMTTimeStamp GMTTIMESTAMP;
SET MyGetGMTTimeStamp = GMTTIMESTAMP '1999-12-31 23:59:59.999999';

```

The PutTime reported by WebSphere MQ on z/OS and other times or timestamps can be inconsistent if the CVT field is not set correctly. For details about when this problem can occur, and how to solve it, see The PutTime that is reported by WebSphere MQ on z/OS, and other times or timestamps are inconsistent.

ESQL INTERVAL data type

The INTERVAL data type holds an interval of time. It has a number of subtypes:

- YEAR
- YEAR TO MONTH
- MONTH
- DAY
- DAY TO HOUR
- DAY TO MINUTE
- DAY TO SECOND
- HOUR
- HOUR TO MINUTE
- HOUR TO SECOND
- MINUTE
- MINUTE TO SECOND
- SECOND

All these subtypes describe intervals of time and all can take part in the full range of operations of the INTERVAL type; for example, addition and subtraction operations with values of type DATE, TIME, or TIMESTAMP.

Use the CAST function to convert from one subtype to another, except for intervals described in years and months, or months, which cannot be converted to those described in days, hours, minutes, and seconds.

The split between months and days arises because the number of days in each month varies. An interval of one month and a day is not meaningful, and cannot be sensibly converted into an equivalent interval in numbers of days only.

An interval literal is defined by the syntax:

```
INTERVAL <interval string> <interval qualifier>
```

The format of interval string and interval qualifier are defined by the table below.

Interval qualifier	Interval string format	Example
YEAR	'<year>' or '<sign> <year>'	'10' or '-10'
YEAR TO MONTH	'<year>-<month>' or '<sign> <year>-<month>'	'2-06' or '- 2-06'
MONTH	'<month>' or '<sign> <month>'	'18' or '-18'
DAY	'<day>' or '<sign> <day>'	'30' or '-30'
DAY TO HOUR	'<day> <hour>' or '<sign> <day> <hour>'	'1 02' or '-1 02'
DAY TO MINUTE	'<day> <hour>:<minute>' or '<sign> <day> <hour>:<minute>'	'1 02:30' or '-1 02:30'
DAY TO SECOND	'<day> <hour>:<minute>:<second>' or '<sign> <day> <hour>:<minute>:<second>'	'1 02:30:15' or '-1 02:30:15.333'
HOUR	'<hour>' or '<sign> <hour>'	'24' or '-24'

Interval qualifier	Interval string format	Example
HOUR TO MINUTE	'<hour>:<minute>' or '<sign> <hour>:<minute>'	'1:30' or '-1:30'
HOUR TO SECOND	'<hour>:<minute>:<second>' or '<sign> <hour>:<minute>:<second>'	'1:29:59' or '-1:29:59.333'
MINUTE	'<minute>' or '<sign> <minute>'	'90' or '-90'
MINUTE TO SECOND	'<minute>:<second>' or '<sign> <minute>:<second>'	'89:59' or '-89:59'
SECOND	'<second>' or '<sign> <second>'	'15' or '-15.7'

Where an interval contains both a year and a month value, a hyphen is used between the two values. In this instance, the month value must be within the range [0, 11]. If an interval contains a month value and no year value, the month value is unconstrained.

A space is used to separate days from the rest of the interval.

If an interval contains more than one of HOUR, MINUTE, and SECOND, a colon is needed to separate the values and all except the leftmost are constrained as follows:

HOUR

0-23

MINUTE

0-59

SECOND

0-59.999...

|

The largest value of the left-most value in an interval is +/- 2147483647.

Some examples of valid interval values are:

- 72 hours
- 3 days: 23 hours
- 3600 seconds
- 90 minutes: 5 seconds

Some examples of invalid interval values are:

- 3 days: 36 hours
A day field is specified, so the hours field is constrained to [0,23].
- 1 hour: 90 minutes
An hour field is specified, so minutes are constrained to [0,59].

Here are some examples of interval literals:

```
INTERVAL '1' HOUR
INTERVAL '90' MINUTE
INTERVAL '1-06' YEAR TO MONTH
```

Representation of ESQL datetime data types

When your application sends a message to a broker, the way in which the message data is interpreted depends on the content of the message itself and the configuration of the message flow. If your application sends a message to be interpreted either by the generic XML parser, or the MRM parser, that is tailored by an XML physical format, the application can include date or time data that is represented by any of the XML schema primitive datetime data types.

The XML schema data type of each piece of data is converted to an ESQL data type, and the element that is created in the logical message tree is of the converted type. If the datetime data in an input message does not match the rules of the chosen schema data type, the values that the parser writes to the logical message tree are modified even if the message is in the MRM domain and you have configured the message flow to validate the input message. (Validation is not available for generic XML messages.)

This has the following effect on the subfields of the input datetime data:

- If any of the subfields of the input message are missing, a default value is written to the logical message tree. This default is substituted from the full timestamp that refers to the beginning of the current epoch: 1970-01-01 00:00:00.
- If the input message contains information for subfields that are not present in the schema, the additional data is discarded. If this occurs, no exception is raised, even if a message in the MRM domain is validated.
- After the data is parsed, it is cast to one of three ESQL datetime data types. These are DATE, TIME, and TIMESTAMP.
 - If a datetime value contains only date subfields, it is cast to an ESQL DATE.
 - If a datetime value contains only time subfields, it is cast to an ESQL TIME.
 - If a datetime value contains both date and time subfields, it is cast to an ESQL TIMESTAMP.

The following examples illustrate these points.

Input data XML schema data type	Schema rules	Input value in the bit stream	Value written to the logical tree (ESQL data type in brackets)
xsd:dateTime	CCYY-MM-DDThh:mm:ss	2002-12-31T23:59:59	2002-12-31 23:59:59 (TIMESTAMP)
		--24	1970-01-24 (DATE)
		23:59:59	23:59:59 (TIME)
xsd:date	CCYY-MM-DD	2002-12-31	2002-12-31 (DATE)
		2002-12-31T23:59:59	2002-12-31 (DATE)
		-06-24	1970-06-24 (DATE)
xsd:time	hh:mm:ss	14:15:16	14:15:16 (TIME)
xsd:gDay	---DD	---24	1970-01-24 (DATE)
xsd:gMonth	--MM	--12	1970-12-01 (DATE)
xsd:gMonthDay	--MM-DD	--12-31	1970-12-31 (DATE)
xsd:gYear	CCYY	2002	2002-01-01 (DATE)
xsd:gYearMonth	CCYY-MM	2002-12	2002-12-01 (DATE)

Validation with missing subfields: When you consider which schema datetime data type to use, bear in mind that, if the message is in the MRM domain, and you configure the message flow to validate messages, missing subfields can cause validation exceptions.

The schema data types Gday, gMonth, gMonthDay, gYear, and gYearMonth are used to record particular recurring periods of time. There is potential confusion when validation is turned on, because the recurring periods of time that are used in these schema data types are stored by ESQL as specific points in time.

For example, when the 24th of the month, which is a gDay (a monthly day) type, is written to the logical tree, the missing month and year subfields are supplied from the epoch (January 1970) to provide the specific date 1970-01-24. If you code ESQL to manipulate this date, for example by adding an interval of 10 days, and then generate an output message that is validated, an exception is raised. This is because the result of the calculation is 1970-02-03 which is invalid because the month subfield of the date no longer matches the epoch date.

ESQL NULL data type

All ESQL data types (except REFERENCE) support the concept of the null value. A value of null means that the value is unknown, undefined, or uninitialized. Null values can arise when you refer to message fields that do not exist, access database columns for which no data has been supplied, or use the keyword NULL, which supplies a null literal value.

Null is a distinct state and is not the same as any other value. In particular, for integers it is not the same thing as the value 0 and for character variables it is not the same thing as a string of zero characters. The rules of ESQL arithmetic take null values into account, and you are typically unaware of their existence. Generally, but not always, these rules mean that, if any operand is null, the result is null.

If an expression returns a null value its data type is not, in general, known. All null values, whatever their origin, are therefore treated equally.

This can be regarded as their belonging to the data type NULL, which is a data type that can have just one value, null.

An expression always returns NULL if any of its elements are NULL.

Testing for null values

To test whether a field contains a null value, use the IS operator described in **Operator=**.

The effect of setting a field to NULL

Take care when assigning a null value to a field. For example, the following command *deletes* the Name field:

```
SET OutputRoot.XML.Msg.Data.Name = NULL; -- this deletes the field
```

The correct way to assign a null value to a field is as follows:

```
SET OutputRoot.XML.Msg.Data.Name VALUE = NULL;  
-- this assigns a NULL value to a field without deleting it
```

ESQL numeric data types

ESQL supports several data types that handle numeric values.

The following data types are collectively known as **numeric** data types:

- “ESQL DECIMAL data type” on page 787
- “ESQL FLOAT data type” on page 788
- “ESQL INTEGER data type” on page 788

Notes:

1. INTEGER and DECIMAL types are represented exactly inside the broker; FLOAT types are inherently subject to rounding error without warning. Do not use FLOAT if you need absolute accuracy, for example, to represent money.
2. Various casts are possible between different numeric types. These can result in loss of precision, if exact types are cast into FLOAT.

For information about numeric functions see “ESQL numeric functions” on page 902.

ESQL DECIMAL data type

The DECIMAL data type holds an exact representation of a decimal number. Decimals have precision, scale, and rounding. Precision is the total number of digits of a number:

- The minimum precision is 1
- The maximum precision is 34

Scale is the number of digits to the right of the decimal point:

- The minimum scale (-exponent) is -999,999,999
- The maximum scale (-exponent) is +999,999,999

You cannot define precision and scale when declaring a DECIMAL, because they are assigned automatically. It is only possible to specify precision and scale when casting to a DECIMAL.

Scale, precision, and rounding:

The following scale, precision, and rounding rules apply:

- Unless rounding is required to keep within the maximum precision, the scale of the result of an addition or subtraction is the greater of the scales of the two operands.
- Unless rounding is required to keep within the maximum precision, the scale of the result of a multiplication is the sum of the scales of the two operands.
- The precision of the result of a division is the smaller of the number of digits needed to represent the result exactly and the maximum precision.
- All addition, subtraction, multiplication, and division calculations round the least significant digits, as necessary, to stay within the maximum precision
- All automatic rounding is *banker's* or *half even symmetric* rounding. The rules of this are:
 - When the first dropped digit is 4 or less, the first retained digit is unchanged
 - When the first dropped digit is 6 or more, the first retained digit is incremented
 - When the first dropped digit is 5, the first retained digit is incremented if it is odd, and unchanged if it is even. Therefore, both 1.5 and 2.5 round to 2 while 3.5 and 4.5 both round to 4
 - Negative numbers are rounded according to the same rule

Decimal literals:

Decimal literals that consist of an unquoted string of digits only, that is, that contain neither a decimal point nor an exponent (for example 12345) are of type INTEGER if they are small enough to be represented as integers. Otherwise they are of type DECIMAL.

Decimal literals that consist of an unquoted string of digits, optionally a decimal point, and an exponent (for example 123e1), are of type FLOAT if they are small enough to be represented as floats. Otherwise they are of type DECIMAL.

Decimal literals that consist of the keyword DECIMAL and a quoted string of digits, with or without a decimal point and with or without an exponent, are of type DECIMAL, for example, DECIMAL '42', DECIMAL '1.2346789e+203'.

The strings in this type of literal can also have the values:

- 'NAN', not a number
- 'INF', 'INFINITY'
- '+INF', '+INFINITY'
- '-INF', '-INFINITY'
- 'MAX'
- 'MIN'

(in any mixture of case) to denote the corresponding values

ESQL FLOAT data type

The FLOAT data type holds a 64-bit, base 2, fraction and exponent approximation to a real number. This gives a range of values between $+1.7E-308$ and $+1.7E+308$.

Float literals consist of an unquoted string of digits and either a decimal point (for example 123.4) or an exponent (for example 123e4) or both (for example 123.4e5). They are of type FLOAT if they are small enough to be represented as floats. Otherwise they are of type DECIMAL.

Rounding:

When you CAST a FLOAT to an INTEGER, either implicitly or explicitly, the FLOAT is truncated; that is, the numbers after the decimal point are removed and no rounding occurs.

ESQL INTEGER data type

The INTEGER data type holds an integer number in 64-bit two's complement form. This gives a range of values between -9223372036854775808 and $+9223372036854775807$.

Integer literals consist of an unquoted string of digits only; that is, they contain neither a decimal point nor an exponent; for example, 12345. They are of type INTEGER if they are small enough to be represented as integers. Otherwise they are of type DECIMAL.

In addition to this format, you can write integer literals in hexadecimal notation; for example, 0x1234abcd. You can write the hexadecimal letters A to F, and the "x" after the initial zero, in uppercase or lowercase. If you use hexadecimal format, the number must be small enough to fit into an integer. (That is, it cannot be a decimal.)

ESQL REFERENCE data type

The REFERENCE data type holds the location of a field in a message. It cannot hold the location of a constant, a database table, a database column, or another reference.

Note: For backward compatibility, reference variables can also point at scalar variables

A reference literal is an hierarchic path name, consisting of a list of path elements separated by periods. The first element in the list is known as the correlation name, and identifies a reference, row, or scalar variable. Any subsequent elements apply to references to message trees only, and identify field types, names, and indexes within the message tree relative to the field pointed to by the correlation name.

For example:

```
InputRoot.MQMD.Priority
```

is a field reference literal that refers to the Priority field contained within an MQMD structure within an input message.

ESQL string data types

ESQL supports several data types that handle string values. The following data types are collectively known as **string** data types:

- “ESQL BIT data type”
- “ESQL BLOB data type”
- “ESQL CHARACTER data type” on page 790

For information about string functions, see “ESQL string manipulation functions” on page 915.

ESQL BIT data type

The BIT data type holds a variable length string of binary digits. It is commonly used to represent arbitrary binary data that does not contain an exact number of bytes. A bit string literal consists of the letter B, followed by a string of binary digits enclosed in single quotation marks, as in the following example:

```
B'0100101001'
```

Any number of digits, which must be either 0 or 1, can be specified. The initial B can be specified in uppercase or lowercase.

ESQL BLOB data type

The BLOB data type holds a variable length string of 8-bit bytes. It is commonly used to represent arbitrary binary data. A BLOB literal consists of the letter X, followed by a string of hexadecimal digits enclosed in single quotation marks, as in the following example:

```
X'0123456789ABCDEF'
```

There must be an even number of digits in the string, because two digits are required to define each byte. Each digit can be one of the hexadecimal digits 0-9 and A-F. Both the initial X and the hexadecimal letters can be specified in uppercase or lowercase.

ESQL CHARACTER data type

The character data type holds a variable length string of Unicode characters. A character string literal consists of any number of characters in single quotation marks. If you want to include a single quotation mark within a character string literal, use another single quotation mark as an escape character.

For example, the assignment SET X='he''was'' puts the value he'was' into X.

ESQL-to-Java data-type mapping table

The following table summarizes the mappings from ESQL to Java.

Notes:

- Only the Java scalar wrappers are passed to Java.
- The ESQL scalar types are mapped to Java data types as object wrappers, or object wrapper arrays, depending upon the direction of the procedure parameter. Each wrapper array contains exactly one element.
- Scalar object wrappers are used to allow NULL values to be passed to and from Java methods.

ESQL data types ¹	Java IN data types	Java INOUT and OUT data types
INTEGER, INT	java.lang.Long	java.lang.Long []
FLOAT	java.lang.Double	java.lang.Double[]
DECIMAL	java.math.BigDecimal	java.math.BigDecimal[]
CHARACTER, CHAR	java.lang.String	java.lang.String[]
BLOB	byte[]	byte[][]
BIT	java.util.BitSet	java.util.BitSet[]
DATE	com.ibm.broker.plugin.MbDate	com.ibm.broker.plugin.MbDate[]
TIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
GMTTIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
TIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
GMTTIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
INTERVAL	Not supported	Not supported
BOOLEAN	java.lang.Boolean	java.lang.Boolean[]
REFERENCE (to a message tree) ^{3 4} _{5 6}	com.ibm.broker.plugin.MbElement	com.ibm.broker.plugin.MbElement[] (Supported for INOUT. Not supported for OUT)
ROW	Not supported	Not supported
LIST	Not supported	Not supported

1. Variables that are declared to be CONSTANT (or references to variables declared to be CONSTANT) are not allowed to have the direction INOUT or OUT.

2. The time zone set in the Java variable is not important; you obtain the required time zone in the output ESQL.
3. The reference parameter cannot be NULL when passed into a Java method.
4. The reference cannot have the direction OUT when passed into a Java method.
5. If an *MbElement* is passed back from Java to ESQL as an INOUT parameter, it must point to a location in the same message tree as that pointed to by the *MbElement* that was passed into the called Java method.

For example, if an ESQL reference to `OutputRoot.XML.Test` is passed into a Java method as an INOUT *MbElement*, but a different *MbElement* is passed back to ESQL when the call returns, the different element must also point to somewhere in the `OutputRoot` tree.

6. An *MbElement* cannot be returned from a Java method with the RETURNS clause, because no ESQL routine can return a reference. However, an *MbElement* can be returned as an INOUT direction parameter, subject to the conditions described in point 5 above.

A REFERENCE to a scalar variable can be used in the CALL of a Java method, provided that the data type of the variable the reference refers to matches the corresponding data type in the Java program signature.

ESQL variables

Types of variable

You can use the “DECLARE statement” on page 851 to define three types of variable:

External

External variables (defined with the EXTERNAL keyword) are also known as *user-defined properties* (UDPs): see “User-defined properties in ESQL” on page 149. They exist for the entire lifetime of a message flow and are visible to all messages passing through the flow. Their initial values (optionally set by the DECLARE statement) can be modified, at design time, by the Message Flow editor, or, at deployment time, by the BAR editor. Their values cannot be modified by ESQL.

Normal

“Normal” variables have a lifetime of just one message passing through a node. They are visible to that message only. To define a “normal” variable, omit both the EXTERNAL and SHARED keywords.

Shared

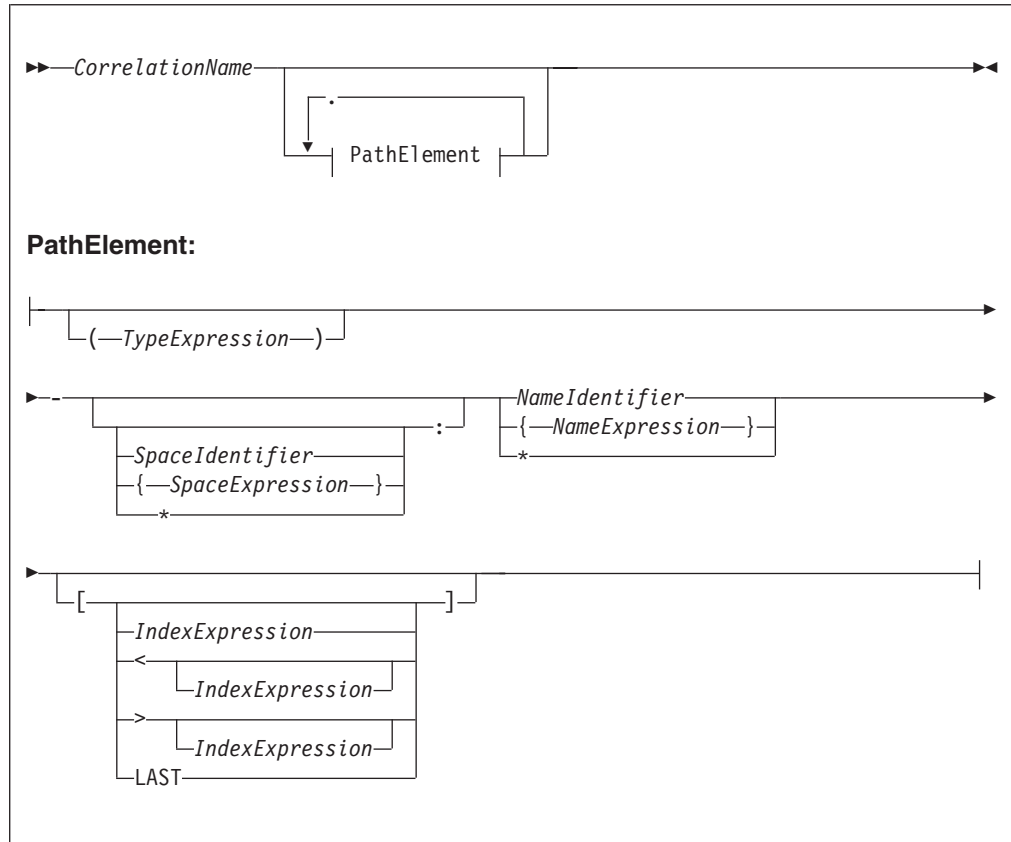
Shared variables can be used to implement an in-memory cache in the message flow, see “Optimizing message flow response times” on page 73. Shared variables have a long lifetime and are visible to multiple messages passing through a flow, see “Long-lived variables” on page 150. They exist for the lifetime of the execution group process, the lifetime of the flow or node, or the lifetime of the node’s SQL that declares the variable (whichever is the shortest). They are initialized when the first message passes through the flow or node after each broker start up.

See also the ATOMIC option of the “BEGIN ... END statement” on page 807. The BEGIN ATOMIC construct is useful when a number of changes need to be made to a shared variable and it is important to prevent other instances seeing the intermediate states of the data.

ESQL field references

This topic describes how to use ESQL field references to form paths to message body elements.

The full syntax for field references is as shown below:



A field reference consists of a correlation name, followed by zero or more path fields separated by periods (.). The correlation name identifies a well-known starting point and must be the name of a constant, a declared variable (scalar, row or reference), or one of the predefined start points; for example, `InputRoot`. The path fields define a path from the start point to the desired field.

For example:

```
InputRoot.XML.Data.Invoice
```

starts the broker at the location `InputRoot` (that is, the root of the input message to a Compute node) and then performs a sequence of navigations. First, it navigates from root to the first child field called `XML`, then to the first child field of the `XML` field called `Data`. Finally, the broker navigates to the first child field of the `Data` field called `Invoice`. Whenever this field reference occurs in an ESQL program, the invoice field is accessed.

This form of field reference is simple, convenient, and is the most commonly used. However, it does have two limitations:

- Because the names used must be valid ESQL identifiers, you can use only names that conform to the rules of ESQL. That is, the names can contain only

alphanumeric characters including underscore, the first character cannot be numeric, and names must be at least one character long. You can avoid these limitations by enclosing names not conforming to these rules in double quotation marks. For example:

```
InputRoot.XML."Customer Data".Invoice
```

If you need to refer to fields that contain quotation marks, use two pairs of quotation marks around the reference. For example:

```
Body.Message. ""hello""
```

Some identifiers are reserved as keywords but, with the exception of the correlation name, you can use them in field references without the use of double quotation marks

- Because the names of the fields appear in the ESQL program, they must be known when the program is written. This limitation can be avoided by using the alternative syntax that uses braces ({ ... }). This syntax allows you to use any expression that returns a non-null value of type character.

For example:

```
InputRoot.XML."Customer Data".{'Customer-' ||  
CurrentCustomer}.Invoice
```

in which the invoices are contained in a folder with a name is formed by concatenating the character literal Customer- with the value in CurrentCustomer (which in this example must be a declared variable of type character).

You can use the asterisk (*) wildcard character in a path element to match any name. You can also use "*" to specify a partial name. For example, Prefix* matches any name that begins with "Prefix".

Note that enclosing anything in double quotation marks in ESQL makes it an identifier; enclosing anything in single quotation marks makes it a character literal. You must enclose all character strings in single quotation marks.

See:

- "Namespaces" for the meaning of the different combinations of namespace and name
- "Target field references" on page 796 for the meaning of the different combinations of field references
- "Indexes" on page 794 for the meaning of the different combinations of index clauses
- "Types" on page 794 for the meaning of the different combinations of types

Namespaces

Field names can belong to namespaces. Field references provide support for namespaces as follows:

- Each field of each field reference that contains a name clause can also contain a namespace clause defining the namespace to which the specified name belongs.
- Each namespace name can be defined by either a simple identifier or by an expression (enclosed in curly braces). If an identifier is the name of a declared namespace constant, the value of the constant is used. If an expression is used, it must return a non-null value of type character.
- A namespace clause of * explicitly states that namespace information is to be ignored when locating Fields in a tree.

- A namespace clause with no identifier, expression, or *, that is, only the : present, explicitly targets the notarget namespace

Indexes

Each field of a field reference can contain an index clause. This clause is denoted by brackets ([...]) and accepts any expression that returns a non-null value of type integer. This clause identifies which of several fields with the same name is to be selected. Fields are numbered from the first, starting at one. If this clause is not present, it is assumed that the first field is required. Thus, the two examples below have exactly the same meaning:

```
InputRoot.XML.Data[1].Invoice
InputRoot.XML.Data.Invoice[1]
```

This construct is most commonly used with an index variable, so that a loop steps through all such fields in sequence. For example:

```
WHILE count < 32 DO
    SET TOTAL = TOTAL + InputRoot.XML.Data.Invoice[count].Amount;
    SET COUNT = COUNT + 1
END WHILE;
```

Use this kind of construct with care, because it implies that the broker must count the fields from the beginning each time round the loop. If the repeat count is large, performance will be poor. In such cases, a better alternative is to use a field reference variable.

Index expressions can optionally be preceded by a less-than sign (<), indicating that the required field is to be indexed from the last field, not the first. In this case, the index 1 refers to the last field and the index 2 refers to the penultimate field. For completeness, you can use a greater-than sign to indicate counting from the first field. The example below shows ESQL code that handles indexes where there are four fields called Invoice.

```
InputRoot.XML.Data.Invoice      -- Selects the first
InputRoot.XML.Data.Invoice[1]   -- Selects the first
InputRoot.XML.Data.Invoice[>]  -- Selects the first
InputRoot.XML.Data.Invoice[>1] -- Selects the first
InputRoot.XML.Data.Invoice[>2] -- Selects the second
InputRoot.XML.Data.Invoice[<]  -- Selects the fourth
InputRoot.XML.Data.Invoice[<1] -- Selects the fourth
InputRoot.XML.Data.Invoice[<2] -- Selects the third
InputRoot.XML.Data.Invoice[<3] -- Selects the second
```

An index clause can also consist of an empty pair of brackets ([]). This selects all fields with matching names. Use this construct with functions and statements that expect lists (for example, the SELECT, CARDINALITY, SINGULAR, and EXISTS functions, or the SET statement) .

Types

Each field of a field reference can contain a type clause. These are denoted by parentheses (()), and accept any expression that returns a non-null value of type integer. The presence of a type expression restricts the fields that are selected to those of the matching type. This construct is most commonly used with generic XML, where there are many field types and it is possible for one XML field to contain both attributes and further XML Fields with the same name.

For example:

```
<Item Value = '1234' >
  <Value>5678</Value>
</Item>
```

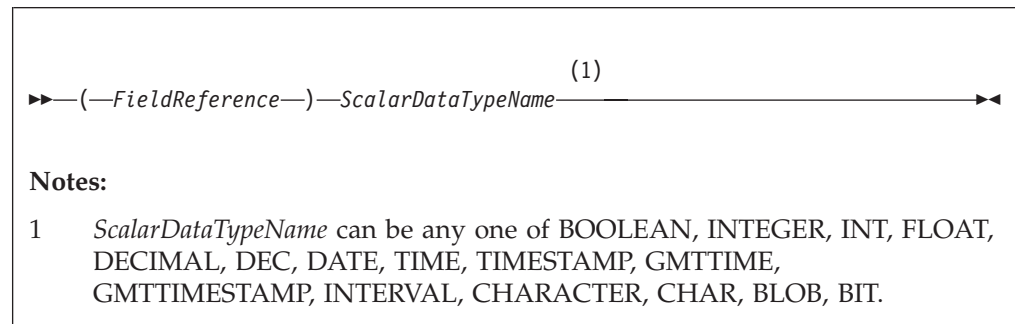
Here, the XML field Item has two child Fields, both called “Value”. The child Fields can be distinguished by using type clauses:

Item.(<Domain>.Attribute)Value to select the attribute, and

Item.(XML.Element)Value to select the field, where <Domain> is one of XML, XMLNS, or XMLNSC, as determined by the message domain of the source.

Type constraints

A type constraint checks the data type returned by a field reference.



Typically, a type constraint causes the scalar value of the reference to be extracted (in a similar way to the FIELDVALUE function) and an exception to be thrown if the reference is not of the correct type. By definition, an exception will be thrown for all nonexistent fields, because these evaluate to NULL. This provides a convenient and fast way of causing exceptions if essential fields are missing from messages.

However, when type constraints occur in expressions that are candidates for being passed to a database (for example, they are in a WHERE clause), the information is used to determine whether the expression can be given to the database. This can be important if a WHERE clause contains a CAST operating on a database table column. In the absence of a type constraint, such expressions cannot be given to the database because the broker cannot tell whether the database is capable of performing the required conversion. Note, however, that you should always exercise caution when using casts operating on column values, because some databases have exceedingly limited data conversion capabilities.

Summary

*****, ***[.]**, **(.)***, **(.)*[.]**

None of these forms specifies a name or namespace. The target field can have any name, in any namespace or in no namespace. It is located solely by its type, its index, or its type and index, as appropriate.

NameId, **NameId[.]**, **(.)NameId**, **(.)NameId[.]**

All these forms specify a name but no namespace. The target field is located by namespace and name, and also by type and index where appropriate.

The namespace is taken to be the only namespace in the namespace path containing this name. The only namespace that can be in the path is the notarget namespace.

These forms all existed before namespaces were introduced. Although their behavior has changed in that they now compare both name and namespace, existing transforms should see no change in their behavior because all existing transforms create their Fields in the notarget namespace.

: *, **:[*..]**, **(.:)***, **(.:)*[..]**

All these forms specify the notarget namespace but no name. The target field is located by its namespace and also by type and index where appropriate.

: NameId, **:NameId[..]**, **(.:)NameId**, **(.:)NameId[..]**

All these forms specify a name and the notarget namespace. The target field is located by namespace and name and also by type and index where appropriate.

*** :***, ***:[*..]**, **(.)*:***, **(.)*:[*..]**

None of these forms specifies a name or a namespace. Note that `"*:*"` is equivalent to `"*"`, and matches no namespace as well as any namespace. The target field can have any name, in any namespace or in no namespace. It is located solely by its type, its index, or its type and index, as appropriate.

*** :NameId**, ***:NameId[..]**, **(.)*:NameId**, **(.)*:NameId[..]**

All these forms specify a name but no namespace. The target field is located by name and also by type and index where appropriate.

SpaceId :*, **SpaceId:[*..]**, **(.)SpaceId:***, **(.)SpaceId:[*..]**

All these forms specify a namespace but no name. The target field is located by namespace and also by type and index where appropriate.

SpaceId :NameId, **SpaceId:NameId[..]**, **(.)SpaceId:NameId**, **(.)SpaceId:NameId[..]**

All these forms specify a namespace and name. The target field is located by namespace and name and also by type and index where appropriate.

In all the preceding cases a name, or namespace, provided by an expression contained in braces ({}) is equivalent to a name provided as an identifier.

By definition, the name of the notarget namespace is the empty string. The empty string can be selected by expressions which evaluate to the empty string, the empty identifier `""`, or by reference to a namespace constant defined as the empty string.

Target field references

The use of field references usually implies searching for an existing field. However, if the required field does not exist, as is usually the case for field references that are the targets of SET statements and those in the AS clauses of SELECT functions, it is created.

In these situations, there are a variety of circumstances in which the broker cannot tell what the required name or namespace is, and in these situations the following general principles apply :

- If the name clause is absent or does not specify a name, and the namespace clause is absent or does not specify or imply a namespace (that is, there is no name or namespace available), one of the following conditions applies:

- If the assignment algorithm does **not** copy the name from some existing field, the new field has both its name and namespace set to the empty string and its name flag is **not** set automatically.
In the absence of a type specification, the field’s type is not *Name* or *NameValue*, which effectively indicates that the new field is nameless.
- Otherwise, if the assignment algorithm chooses to copy the name from some existing field, the new field has both its name and namespace copied from the existing field and its *Name* flag is set automatically
- If the name clause is present and specifies a name, but the namespace clause is absent or does not specify or imply a namespace (that is, a name is available but a namespace is not), the new field has its:
 - *Name* set to the given value
 - *Namespace* set to the empty string
 - *Name* flag set automatically
- If the name clause is absent or does not specify a name, but the namespace clause is present and specifies or implies a namespace (that is, a namespace is available but a name is not), the new field has its:
 - *Namespace* set to the given value
 - *Name* set to the empty string
 - *Name* flag set automatically
- If the name clause is present and specifies a name, and the namespace clause is present and specifies or implies a namespace, the new field has its:
 - *Name* set to the given value
 - *Namespace* set to the given value
 - *Name* flag set automatically

There are also cases where the broker creates Fields in addition to those referenced by field references:

- Tree copy: new Fields are created by an algorithm that uses a source tree as a template. If the algorithm copies the name of a source field to a new field, its namespace is copied as well.
- Anonymous select expressions: SELECT clauses are not obliged to have AS clauses; those that do not have them, set the names of the newly created Fields to default values (see “SELECT function” on page 954).

These defaults can be derived from field names, column names or can simply be manufactured sequence names. If the name is an field name, this is effectively a tree copy, and the namespace name is copied as above.

Otherwise, the namespace of the newly-created field is derived by searching the path, that is, the name is be treated as the *NameId* syntax of a field reference.

The effect of setting a field to NULL

Take care when assigning a null value to a field. For example, the following command *deletes* the Name field:

```
SET OutputRoot.XML.Msg.Data.Name = NULL; -- this deletes the field
```

The correct way to assign a null value to a field is as follows:

```
SET OutputRoot.XML.Msg.Data.Name VALUE = NULL;
-- this assigns a NULL value to a field without deleting it
```

Note: to users on backward compatibility

For backward compatibility the LAST keyword is still supported, but its use is deprecated. LAST cannot be used as part of an index expression: [LAST] is valid, and is equivalent to [<], but [LAST3] is not valid.

The LAST keyword has been replaced by the following arrow syntax, which allows both a direction of search and index to be specified:

```
Field [ > ]                -- The first field, equivalent to [ 1 ]
Field [ > (a + b) * 2 ]
Field [ < ]                -- The last field, equivalent to [ LAST ]
Field [ < 1 ]              -- The last field, equivalent to [ LAST ]
Field [ < 2 ]              -- The last but one field
Field [ < (a + b) / 3 ]
```

ESQL operators

This section provides reference information for the following groups of operators, and for the rules for precedence:

- Simple comparison operators
- Complex comparison operators
- Logical operators
- Numeric operators
- String operator
- Rules for operator precedence

ESQL simple comparison operators

This topic describes ESQL's simple comparison operators. For information about ESQL's complex comparison operators, see "ESQL complex comparison operators" on page 799.

ESQL provides a full set of comparison operators (predicates). Each compares two scalar values and returns a Boolean. If either operand is null the result is null. Otherwise the result is true if the condition is satisfied and false if it is not.

Comparison operators can be applied to all scalar data types. However, if the two operands are of different types, special rules apply. These are described in "Implicit casts" on page 973.

Some comparison operators also support the comparison of rows and lists. These are noted below.

Operator >

The first operand is greater than the second.

Operator <

The first operand is less than the second.

Operator >=

The first operand is greater than or equal to the second.

Operator <=

The first operand is less than or equal to the second.

Operator =

The first operand is equal to that of the second.

This operator can also compare rows and lists. See "ROW and LIST comparisons" on page 963 for a description of list and row comparison.

Operator <>

The first operand is not equal to the second.

This operator can also compare rows and lists. See “ROW and LIST comparisons” on page 963 for a description of list and row comparison.

The meanings of “equal”, “less”, and “greater” in this context are as follows:

- For the numeric types (INTEGER, FLOAT, DECIMAL) the numeric values are compared. Thus 4.2 is greater than 2.4 and -2.4 is greater than -4.2.
- For the date/time types (DATE, TIME, TIMESTAMP, GMTTIME, GMTTIMESTAMP but not INTERVAL) a later point in time is regarded as being greater than an earlier point in time. Thus the date 2004-03-31 is greater than the date 1947-10-24.
- For the INTERVAL type, a larger interval of time is regarded as being greater than a smaller interval of time.

For the string types (CHARACTER, BLOB, BIT) the comparison is lexicographic. Starting from the left, the individual elements (each character, byte or bit) are compared. If no difference is found, the strings are equal. If a difference is found, the values are greater if the first different element in the first operand is greater than the corresponding element in the second and less if they are less. In the special case where two strings are of unequal length but equal as far as they go, the longer string is regarded as being greater than the shorter. Thus:

'ABD' is greater than 'ABC'
'ABC' is greater than 'AB'

Trailing blanks are regarded as insignificant in character comparisons. Thus if you want to ensure that two strings are truly equal you need to compare both the strings themselves and their lengths. For example:

'ABC ' is equal to 'ABC'

Note that comparing strings with a length of one is equivalent to comparing individual characters, bytes, or bits. Because ESQL has no single character, byte, or bit data types, it is standard practice to use strings of length one to compare single characters, bytes, or bits.

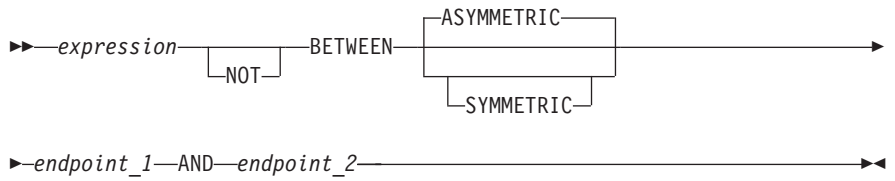
ESQL complex comparison operators

This topic describes ESQL’s complex comparison operators (predicates). For information about ESQL’s simple comparison operators, see “ESQL simple comparison operators” on page 798.

Operator BETWEEN

The operator BETWEEN allows you to test whether a value lies between two boundary values.

BETWEEN operator



This operator exists in two forms, SYMMETRIC and ASYMMETRIC (which is the default if neither is specified). The SYMMETRIC form is equivalent to:

$(\text{source} \geq \text{boundary1} \text{ AND } \text{source} \leq \text{boundary2}) \text{ OR}$
 $(\text{source} \geq \text{boundary2} \text{ AND } \text{source} \leq \text{boundary1})$

The ASYMMETRIC form is equivalent to:
 $\text{source} \geq \text{boundary1} \text{ AND } \text{source} \leq \text{boundary2}$

The ASYMMETRIC form is simpler but returns only the result that you expect when the first boundary value has a smaller value than the second boundary. It is only useful when the boundary condition expressions are literals.

If the operands are of different types, special rules apply. These are described in “Implicit casts” on page 973.

Operator EXISTS

EXISTS operator



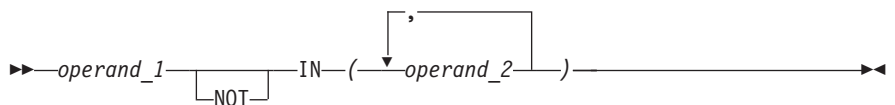
The operator EXISTS returns a boolean value indicating whether a SELECT function returned one or more values (TRUE) or none (FALSE).

$\text{EXISTS}(\text{SELECT } * \text{ FROM something WHERE predicate})$

Operator IN

The operator IN allows you to test whether a value is equal to one of a list of values.

IN operator

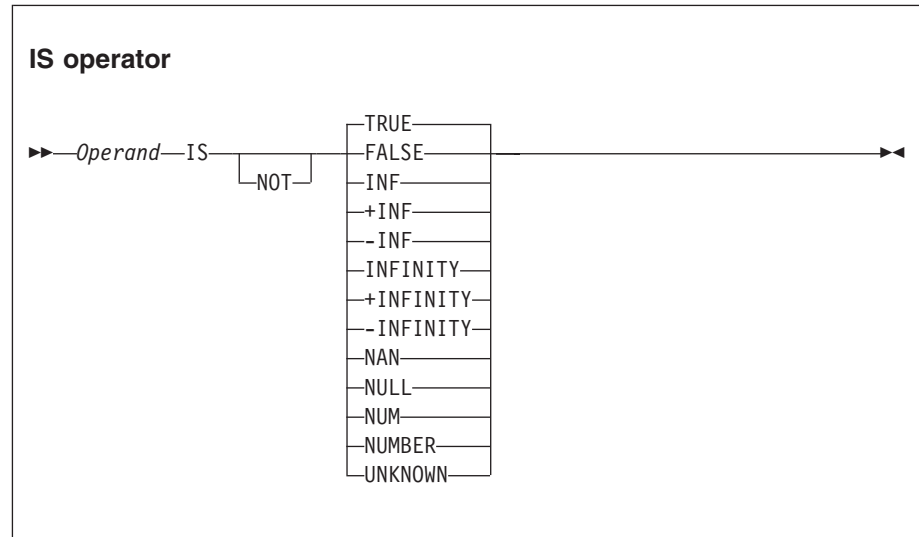


The result is TRUE if the left operand is not NULL and is equal to one of the right operands. The result is FALSE if the left operand is not NULL and is not equal to any of the right operands, none of which have NULL

values. Otherwise the result is UNKNOWN. If the operands are of different types, special rules apply. These are described in “Implicit casts” on page 973.

Operator IS

The operator IS allows you to test whether an expression has returned a special value.



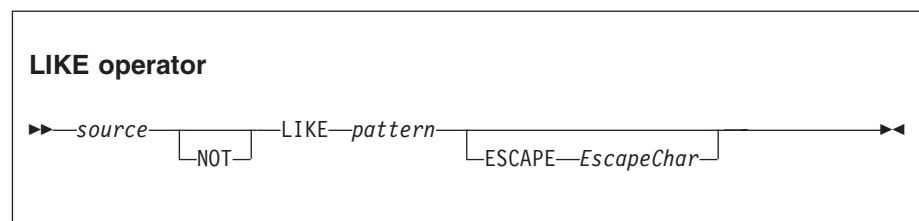
The primary purpose of the operator IS is to test whether a value is NULL. The comparison operator (=) does not allow this because the result of comparing anything with NULL is NULL.

IS also allows you to test for the Boolean values TRUE and FALSE, and the testing of decimal values for special values. These are denoted by INF, +INF, -INF, NAN (not a number), and NUM (a valid number) in any mixture of case. The alternative forms +INFINITY, -INFINITY, and NUMBER are also accepted.

If applied to non-numeric types, the result is FALSE.

Operator LIKE

The operator LIKE searches for strings that match a certain pattern.



The result is TRUE if none of the operands is NULL and the *source* operand matches the *pattern* operand. The result is FALSE if none of the operands is NULL and the *source* operand does not match the *pattern* operand. Otherwise the result is UNKNOWN.

The pattern is specified by a string in which the percent (%) and underscore (_) characters have a special meaning:

- The underscore character _ matches any single character.

For example, the following finds matches for IBM and for IGI, but not for International Business Machines or IBM Corp:

```
Body.Trade.Company LIKE 'I__'
```

- The percent character % matches a string of zero or more characters. For example, the following finds matches for IBM, IGI, International Business Machines, and IBM Corp:

```
Body.Trade.Company LIKE 'I%'
```

To use the percent and underscore characters within the expressions that are to be matched, precede the characters with an ESCAPE character, which defaults to the backslash (\) character.

For example, the following predicate finds a match for IBM_Corp.

```
Body.Trade.Company LIKE 'IBM\_Corp'
```

You can specify a different escape character by using the ESCAPE clause. For example, you could also specify the previous example like this:

```
Body.Trade.Company LIKE 'IBM$_Corp' ESCAPE '$'
```

Operator SINGULAR

SINGULAR operator

▶—*Operand*—(*ListExpression*)—▶

The operator SINGULAR returns a boolean value of TRUE if the list has exactly one element, otherwise it returns FALSE.

ESQL logical operators

ESQL provides the following logical operators:

Operator AND

The result is the logical AND of the two operands. Both operands must be boolean values.

Operator OR

The result is the logical OR of the two operands. Both operands must be boolean values.

Operator NOT

The result is the logical NOT of the operand, which must be a boolean value.

NULL and UNKNOWN values are treated as special values by these operators. The principles are:

- NULL and UNKNOWN are treated the same.
- If an operand is NULL the result is NULL unless the operation result is already dictated by the other parameter.

The result of AND and OR operations is defined by the following table.

Value of P	Value of Q	Result of P AND Q	Result of P OR Q
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE

Value of P	Value of Q	Result of P AND Q	Result of P OR Q
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	UNKNOWN	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	TRUE
UNKNOWN	FALSE	FALSE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

The result of NOT operations is defined by the following table.

Operand	Result of NOT
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

ESQL numeric operators

ESQL provides the following numeric operators:

Unary Operator -

The result is the negation of the operand (that is, it has the same magnitude as the operand but the opposite sign). You can negate numeric values (INTEGER, DECIMAL and FLOAT) and intervals (INTERVAL).

Operator +

The result is the sum of the two operands. You can add two numeric values, two intervals, and an interval to a datetime value (DATE, TIME, TIMESTAMP, GMTTIME, and GMTTIMESTAMP).

Operator -

The result is the difference between the two operands. It is possible to:

- Subtract one numeric value from another.
- Subtract one date-time from another. The result is an interval.
- Subtract one interval from another. The result is an interval.
- Subtract an interval from a datetime value. The result is a date-time.

When subtracting one date-time from another, you must indicate the type of interval required. You do this by using a qualifier consisting of parentheses enclosing the expression, followed by an interval qualifier. For example:

```
SET OutputRoot.XML.Data.Age =
  (DATE '2005-03-31' - DATE '1947-10-24') YEAR TO MONTH;
```

Operator *

The result is the product of the two operands. You can multiply numeric values and multiply an interval by a numeric value.

Operator /

The result is the dividend of the two operands. You can divide numeric values and divide an interval by a numeric value.

Operator ||

The result is the concatenation of the two operands. You can concatenate string values (CHARACTER, BIT, and BLOB).

In all cases, if either operand is NULL, the result is NULL. If the operands are of different types, special rules apply. These are described in “Implicit casts” on page 973.

For examples of how you can use these operators to manipulate datetime values, see “Using numeric operators with datetime values” on page 183.

ESQL string operator

ESQL provides the following string operator:

Operator ||

The result is the concatenation of the two operands. You can concatenate string values (CHARACTER, BIT, and BLOB).

If either operand is NULL, the result is NULL.

Rules for ESQL operator precedence

When an expression involves more than one operator, the order in which the expression is evaluated might affect the result. Consider the following example:

```
SET a = b + c * d;
```

Under ESQL’s precedence rules, *c* is multiplied by *d* and the result is added to *b*. This rule states that multiplication takes precedence over addition, so reordering the expression as follows:

```
SET a = c * d + b;
```

makes no difference. ESQL’s precedence rules are set out below but it is generally considered good practice to use parentheses to make the meaning clear. The order of precedence is:

1. Parentheses
2. Unary operators including unary - and NOT
3. Multiplication and division
4. Concatenation
5. Addition and subtraction

Operations at the same level are evaluated from left to right.

ESQL statements

The following table summarizes the ESQL statements and what they do.

Statement type	Description
Basic statements:	
“BEGIN ... END statement” on page 807	Gives the statements defined within the BEGIN and END keywords the status of a single statement.
“CALL statement” on page 813	Invokes a user-written routine that has been defined using a CREATE FUNCTION or CREATE PROCEDURE statement.

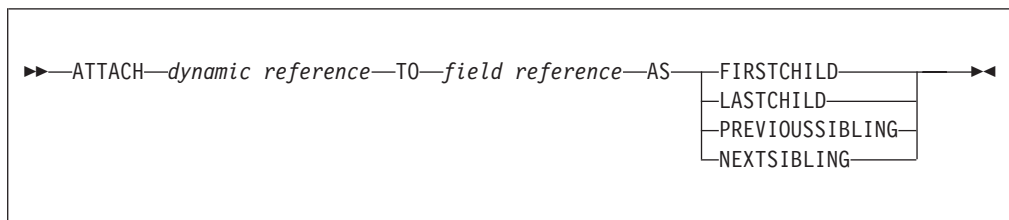
Statement type	Description
“CASE statement” on page 816	Uses rules defined in WHEN clauses to select a block of statements to execute.
“CREATE FUNCTION statement” on page 826	Like CREATE PROCEDURE, CREATE FUNCTION defines a user-written routine. (The few differences between CREATE FUNCTION and CREATE ROUTINE are described in the reference material.)
“CREATE MODULE statement” on page 835	Creates a module (a named container associated with a node).
“CREATE PROCEDURE statement” on page 837	Like CREATE FUNCTION, CREATE PROCEDURE defines a user-written routine. (The few differences between CREATE FUNCTION and CREATE ROUTINE are described in the reference material.)
“DECLARE statement” on page 851	Declares one or more variables that can be used to store temporary values.
“IF statement” on page 863	Processes a set of statements based on the result of evaluating condition expressions.
“ITERATE statement” on page 866	Abandons processing the current iteration of the containing WHILE, REPEAT, LOOP, or BEGIN statement, and might start the next iteration.
“LEAVE statement” on page 866	Abandons processing the current iteration of the containing WHILE, REPEAT, LOOP or BEGIN statement, and stops looping.
“LOOP statement” on page 869	Processes a sequence of statements repeatedly and unconditionally.
“REPEAT statement” on page 877	Processes a sequence of statements and then evaluates a condition expression. If the expression evaluates to TRUE, executes the statements again.
“RETURN statement” on page 878	Stops processing the current function or procedure and passes control back to the caller.
“SET statement” on page 880	Evaluates a source expression, and assigns the result to the target entity.
“THROW statement” on page 884	Generates a user exception.
“WHILE statement” on page 888	Evaluates a condition expression, and if it is TRUE executes a sequence of statements.
Message tree manipulation statements:	
“ATTACH statement” on page 806	Attaches a portion of a message tree into a new position in the message hierarchy.
“CREATE statement” on page 818	Creates a new message field.
“DELETE statement” on page 860	Detaches and destroys a portion of a message tree, allowing its memory to be reused.
“DETACH statement” on page 860	Detaches a portion of a message tree without deleting it.
“FOR statement” on page 862	Iterates through a list (for example, a message array).

Statement type	Description
"MOVE statement" on page 870	Changes the field pointed to by a target reference variable.
Database update statements:	
"DELETE FROM statement" on page 857	Deletes rows from a table in an external database based on a search condition.
"INSERT statement" on page 864	Adds a new row to an external database.
"PASSTHRU statement" on page 872	Takes a character value and passes it as an SQL statement to an external database.
"UPDATE statement" on page 885	Updates the values of specified rows and columns in a table in an external database.
Node interaction statements:	
"PROPAGATE statement" on page 874	Propagates a message to the downstream nodes within the message flow.
Other statements:	
"BROKER SCHEMA statement" on page 810	This statement is optional and is used in an ESQL file to explicitly identify the schema that contains the file.
"DECLARE HANDLER statement" on page 856	Declares an error handler.
"EVAL statement" on page 861	Takes a character value, interprets it as an SQL statement, and executes it.
"LOG statement" on page 867	Writes a record to the event or user trace log.
"RESIGNAL statement" on page 878	Re-throws the current exception (if any). This is used by an error handler, when it cannot handle an exception, to give an error handler in higher scope the opportunity of handling the exception.

ATTACH statement

The ATTACH statement attaches a portion of a message tree into a new position in the message hierarchy.

Syntax



The following example illustrates how to use the ATTACH statement, together with the DETACH statement described in "DETACH statement" on page 860, to modify a message structure. The dynamic reference supplied to the DETACH statement must point to a modifiable message tree such as Environment, LocalEnvironment, OutputRoot, OutputExceptionList, or InputLocalEnvironment.

There are some limitations on the use of ATTACH. In general, elements detached from the output trees of a Compute node are not attached to the environment or to input trees.

For example, if you take the following message:

```
<Data>
  <Order>
    <Item>cheese
      <Type>stilton</Type>
    </Item>
    <Item>bread</Item>
  </Order>
  <Order>
    <Item>garlic</Item>
    <Item>wine</Item>
  </Order>
</Data>
```

the following ESQL statements:

```
SET OutputRoot = InputRoot;
DECLARE ref1 REFERENCE TO OutputRoot.XML.Data.Order[1].Item[1];
DETACH ref1;
ATTACH ref1 TO OutputRoot.XML.Data.Order[2] AS LASTCHILD;
```

result in the following new message structure:

```
<Data>
  <Order>
    <Item>bread</Item>
  </Order>
  <Order>
    <Item>garlic</Item>
    <Item>wine</Item>
    <Item>cheese
      <Type>stilton</Type>
    </Item>
  </Order>
</Data>
```

For information about dynamic references see “Creating dynamic field references” on page 179.

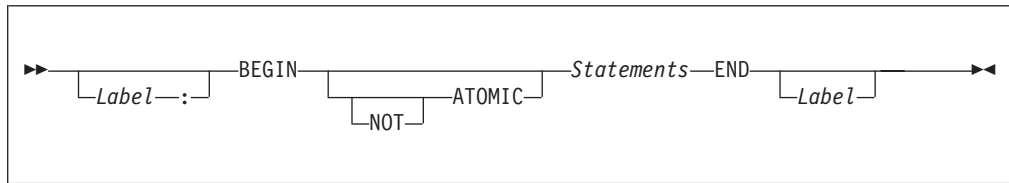
BEGIN ... END statement

The BEGIN ... END statement gives the statements defined within the BEGIN and END keywords the status of a single statement.

This allows the contained statements to:

- Be the body of a function or a procedure
- Have their exceptions handled by a handler
- Have their execution discontinued by a LEAVE statement

Syntax



The second *Label* can be present only if the first *Label* is present. If both labels are present, they must be identical. Two or more labeled statements at the same level can have the same label, but this partly negates the advantage of the second label. The advantage is that the labels unambiguously and accurately match each END with its BEGIN. However, a labeled statement nested within *Statements* cannot have the same label, because this makes the behavior of the ITERATE and LEAVE statements ambiguous.

Scope of variables

A new local variable scope is opened immediately after the opening BEGIN and, therefore, any variables declared within this statement go out of scope when the terminating END is reached. If a local variable has the same name as an existing variable, any references to that name that occur after the declaration access the local variable. For example:

```
DECLARE Variable1 CHAR 'Existing variable';

-- A reference to Variable1 here returns 'Existing variable'

BEGIN
  -- A reference to Variable1 here returns 'Existing variable'

  DECLARE Variable1 CHAR 'Local variable'; -- Perfectly legal even though
  the name is the same

  -- A reference to Variable1 here returns 'Local variable'
END;
```

ATOMIC

If ATOMIC is specified, only one instance of a message flow (that is, one thread) is allowed to execute the statements of a specific BEGIN ATOMIC... END statement (identified by its schema and label), at any one time. If no label is present, the behavior is as if a zero length label had been specified.

The BEGIN ATOMIC construct is useful when a number of changes need to be made to a shared variable and it is important to prevent other instances seeing the intermediate states of the data. Consider the following code example:

```
CREATE PROCEDURE WriteSharedVariable1(IN NewValue CHARACTER)
SharedVariableMutex1 : BEGIN ATOMIC
  -- Set new value into shared variable
END;

CREATE FUNCTION ReadSharedVariable1() RETURNS CHARACTER
SharedVariableMutex1 : BEGIN ATOMIC
  DECLARE Value CHARACTER;
  -- Get value from shared variable
  RETURN Value;
END;
```


The last example assumes that the procedure `WriteSharedVariable1` and the function `ReadSharedVariable1` are in the same schema and are used by nodes within the same flow. However, it does not matter whether or not the procedure and function are contained within modules, or whether they are used within the same or different nodes. The broker ensures that, at any particular time, only one thread is executing any of the statements within the atomic sections. This ensures that, for example, two simultaneous writes or a simultaneous read and write are executed serially. Note that:

- The serialization is limited to the flow. Two flows which use `BEGIN ATOMIC... END` statements with the same schema and label can be executed simultaneously. In this respect, multiple instances within a flow and multiple copies of a flow are not equivalent.
- The serialization is limited by the schema and label. Atomic `BEGIN ... END` statements specified in different schemas or with different labels do not interact with each other.

Note: You can look at this in a different way, if you prefer. For each combination of message flow, schema, and label, the broker has a mutex that prevents simultaneous access to the statements associated with that mutex.

Do not nest `BEGIN ATOMIC... END` statements, either directly or indirectly, because this could lead to “deadly embraces”. For this reason, do not use a `PROPAGATE` statement from within an atomic block.

It is not necessary to use the `BEGIN ATOMIC` construct in flows that will never be deployed with more than one instance (but it might be unwise to leave this to chance). It is also unnecessary to use the `BEGIN ATOMIC` construct on reads and writes to shared variables. The broker always safely writes a new value to, and safely reads the latest value from, a shared variable. `ATOMIC` is only required when the application is sensitive to seeing intermediate results.

Consider the following example:

```
DECLARE LastOrderDate SHARED DATE;
...
SET LastOrderDate = CURRENT_DATE;
...
SET OutputRoot.XMLNSC.Data.Orders.Order[1].Date = LastOrderDate;
```

Here we assume that one thread is periodically updating `LastOrderDate` and another is periodically reading it. There is no need to use `ATOMIC`, because the second `SET` statement always reads a valid value. If the updating and reading occur very closely in time, whether the old or new value is read is indeterminate, but it is always one or the other. The result will never be garbage.

But now consider the following example:

```
DECLARE Count SHARED INT;
...
SET Count = Count + 1;
```

Here we assume that several threads are periodically executing the `SET` statement. In this case you do need to use `ATOMIC`, because two threads might read `Count` in almost the same instant, and get the same value. Both threads perform the addition and both store the same value back. The end result is thus `N+1` and not `N+2`.

The broker does not automatically provide higher-level locking than this (for example, locking covering the whole SET statement), because such locking is liable to cause “deadly embraces”.

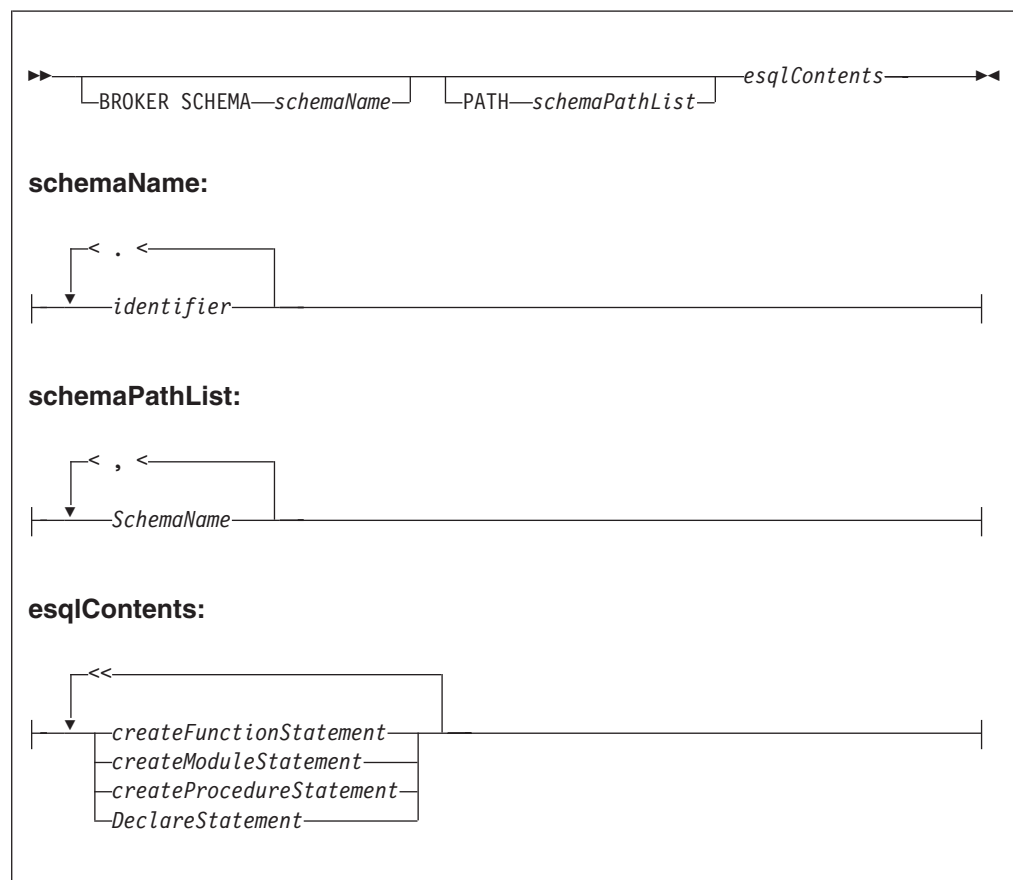
Hint

You can consider the BEGIN ... END statement to be a looping construct, which always loops just once. The effect of an ITERATE or LEAVE statement nested within a BEGIN ... END statement is then as you would expect: control is transferred to the statement following the END. Using ITERATE or LEAVE within a BEGIN ... END statement is useful in cases where there is a long series of computations that needs to be abandoned, either because a definite result has been achieved or because an error has occurred.

BROKER SCHEMA statement

The BROKER SCHEMA statement is optional; use it in an ESQL file to explicitly identify the schema that contains the file.

Syntax



An ESQL schema is a named container for functions, procedures, modules, and variables. It is similar to the namespace concept of C++ and XML, and to the package concept of Java.

In the absence of a BROKER SCHEMA statement, all functions, procedures, modules, and constants belong to the default schema. This is similar to the default namespace in C++, the no-target namespace in XML schema, and the default package in Java.

Note: The BROKER SCHEMA feature is used in the Eclipse tooling set and is a language feature, as is package. It does not appear in the broker ESQL.

PATH clause

The PATH clause specifies a list of additional schemas to be searched when matching function and procedure calls to their implementations. The schema in which the call lies is implicitly included in the PATH.

The PATH feature is used to resolve unqualified function and procedure names in the tools according to the following algorithm.

There must be a single function or procedure that matches the unqualified name, or the tools report an error. You can correct the error by qualifying the function or procedure name with a *schemaId*:

1. The current MODULE (if any) is searched for a matching function or procedure. MODULE-scope functions and procedures are visible only within their containing MODULE. MODULE-scope functions and procedures hide schema-scope functions and procedures.
2. The <node schema> (but none of its contained MODULEs) and the <SQL-broker schema> or schemas identified by the PATH statement are searched for a matching function procedure

Note: The *schemaId* must be a fully qualified schema name.

The <node schema> is the schema containing the node's message flow. The name of this schema is given by the last segment of the message processing node *uuid* in the broker XML message.

When a routine is invoked, the name used can be qualified by the schema name. The behavior depends on the circumstances as follows:

- If the schema is specified, the named schema routine is invoked. The scalar built-in functions, excluding CAST, EXTRACT and the special registers, are considered to be defined within an implicitly declared schema called SQL. They can be invoked in this way, for example, SQL.SUBSTRING(...).

Whatever happens next depends on whether the caller is in a module routine or is a schema routine.

For a module routine:

- If the schema is not specified, the calling statement is in a module routine, and a routine of the given name exists in the local module, that local routine is invoked.
- If the schema is not specified, the calling statement is in a module routine, and a routine of the given name does not exist in the local module, all schemas in the schema path are searched for a routine of the same name.

If a matching function exists in one schema, it is used. If a matching function exists in more than one schema a compile time error occurs. If there is no matching function, the schema SQL is searched.

Note: This rule and the preceding rule imply that a local module routine takes priority over a built-in routine of the same name.

For a schema routine:

- If the schema is not specified, the caller is a schema routine, and a routine of the given name exists in the local schema, that local routine is invoked.
- If the schema is not specified, the calling statement is in a schema routine, and a routine of the given name does not exist in the local schema, all schemas in the schema path are searched for a routine of the same name
If a matching function exists in one schema, it is used. If a matching function exists in more than one schema a compile time error occurs. If there is no matching function, the schema SQL is searched.

Note: This rule and the preceding rule imply that a local schema routine takes priority over a built-in routine of the same name.

The <node schema> is defined as the schema containing the node's message flow. The name of this schema is given by the last segment of the message processing node *uuid* in the broker XML message.

The <node schema> is specified in this manner to provide backward compatibility with previous versions of WebSphere Message Broker

When the <node schema> is the only schema referenced, the broker XML message does not include the extra features contained in WebSphere Message Broker V5.0.

Brokers in previous versions of WebSphere Message Broker do not support multiple schemas, for example, subroutine libraries for reuse. To deploy to a broker in a previous version of the product, put all ESQL subroutines in the same schema as the message flow and node that is invoking them.

Eclipse tooling uses WebSphere Message Broker V5.0 ESQL syntax in content assist and source code validation. When generating broker ESQL code, the Eclipse tooling can generate V2.1 style code for backward compatibility.

The broker schema of the message flow must contain, at the schema level, any of the following in its ESQL files:

- A schema level function
- A schema level procedure
- A schema level constant
- A module level constant
- A module level variable

Without the presence of any of the preceding items, the Eclipse tooling generates broker ESQL without MODULE and FUNCTION Main wrappers. This style is accepted by both V2.1 and V5.0 brokers. However, if you use a V2.1 broker, you cannot use any V5.0 syntax in the code, for example, namespace

Functions and procedure names must be unique within their SCHEMA or MODULE.

Examples

The following example adds a path to a schema called CommonUtils:

```
BROKER SCHEMA CommonUtils  
PATH SpecialUtils;  
  
MODULE ....
```

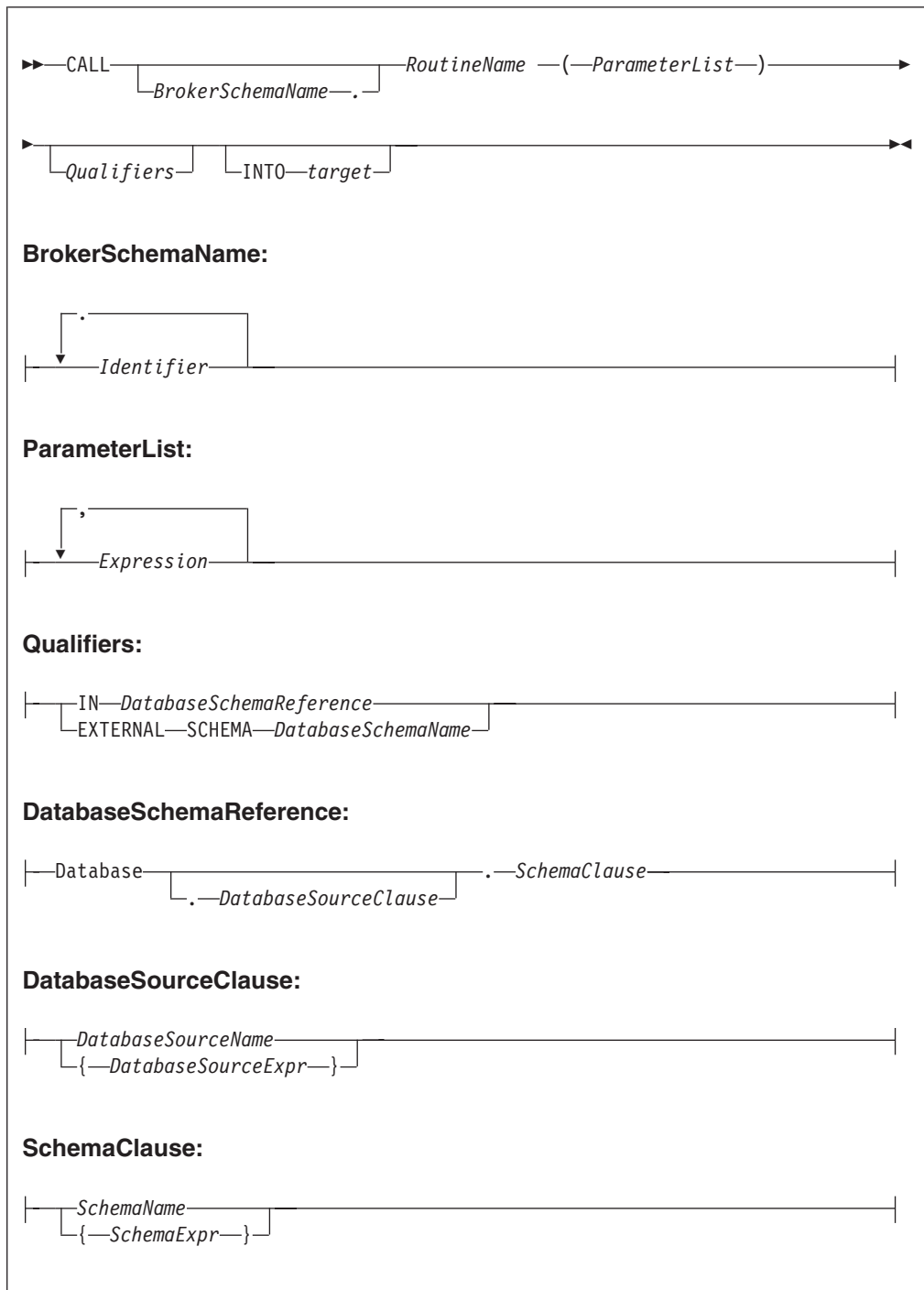
The next example adds a path to the default schema:

```
PATH CommonUtils, SpecialUtils;  
  
MODULE ....
```

CALL statement

The CALL statement calls (invokes) a routine.

Syntax



Using the CALL statement

The CALL statement invokes a routine. A routine is a user-defined function or procedure that has been defined by one of the following:

- A CREATE FUNCTION statement
- A CREATE PROCEDURE statement

Note: As well as standard user-defined functions and procedures, you can also use CALL to invoke built-in (broker-provided) functions and user-defined SQL functions. However, the usual way of invoking these types of function is simply to include their names in expressions.

The called routine must be invoked in a way that matches its definition. For example, if a routine has been defined with three parameters, the first two of type integer and the third of type character, the CALL statement must pass three variables to the routine, each of a data-type that matches the definition. This is called *exact signature matching*, which means that the signature provided by the CALL statement must match the signature provided by the routine's definition.

Exact signature matching applies to a routine's return value as well. If the RETURNS clause is specified on the CREATE FUNCTION statement, or the routine is a built-in function, the INTO clause must be specified on the CALL statement. A return value from a routine cannot be ignored. Conversely, if the RETURNS clause is not specified on the CREATE FUNCTION statement, the INTO clause must not be specified, because there is no return value from the routine.

You can use the CALL statement to invoke a routine that has been implemented in any of the following ways:

- ESQL.
- Java.
- As a stored procedure in a database.
- As a built-in (broker-provided) function. (But see the note above about calling built-in functions.)

This variety of implementation means that some of the clauses in the CALL syntax diagram are not applicable (or allowed) for all types of routine. It also allows the CALL statement to invoke any type of routine, irrespective of how the routine has been defined.

When the optional *BrokerSchemaName* parameter is not specified, the broker SQL parser searches for the named procedure using the algorithm described in the PATH statement (see the "PATH clause" on page 811 of the BROKER SCHEMA statement).

When the *BrokerSchemaName* parameter is specified, the broker SQL parser invokes the named procedure in the specified schema without first searching the path. However, if a procedure reference is ambiguous (that is, there are two procedures with the same name in different broker schemas) and the reference is not qualified by the optional *BrokerSchemaName*, the Eclipse toolset generates a "Tasks view error" that you must correct to deploy the ambiguous code.

The broker-provided built-in functions are automatically placed in a predefined broker schema called SQL. The SQL schema is always searched last for a routine that has not been matched to a user-defined routine. Therefore, a user-defined module takes precedence over a built-in routine of the same name.

Each broker schema provides a unique symbol or namespace for a routine, so a routine name is unique when it is qualified by the name of the schema to which it belongs.

The INTO clause is used to store the return value from a routine that has been defined with a RETURNS clause, or from a built-in function. The *target* can be an

ESQL variable of a data type that matches the data type on the RETURNS clause, or a dot-separated message reference. For example, both of the following ESQL statements are valid:

```
CALL myProc1() INTO cursor;  
CALL myProc1() INTO OutputRoot.XML.TestValue1;
```

The CALL statement passes the parameters into the procedure in the order given to it. Parameters that have been defined as IN or INOUT on the routine's definition are evaluated before the CALL is made, but parameters defined as OUT are always passed in as NULL parameters of the correct type. When the procedure has completed, any parameters declared as OUT or INOUT are updated to reflect any changes made to them during the procedure's execution. Parameters defined as IN are never changed during the course of a procedure's execution.

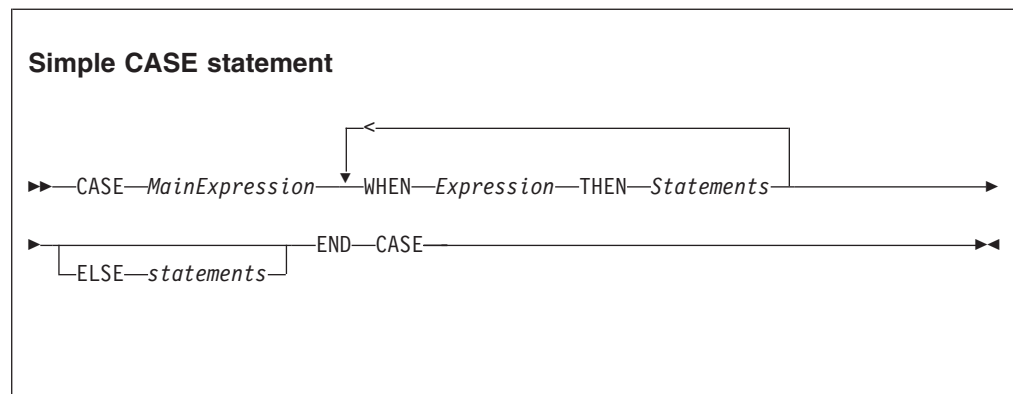
Routine overloading is not supported. This means that you cannot create two routines of the same name in the same broker schema. If the broker detects that a routine has been overloaded, it raises an exception. Similarly, you cannot invoke a database stored procedure that has been overloaded. A database stored procedure is overloaded if another procedure of the same name exists in the same database schema. However, you can invoke an overloaded Java method, as long as you create a separate ESQL definition for each overloaded method you want to call, and give each ESQL definition a unique routine name.

CASE statement

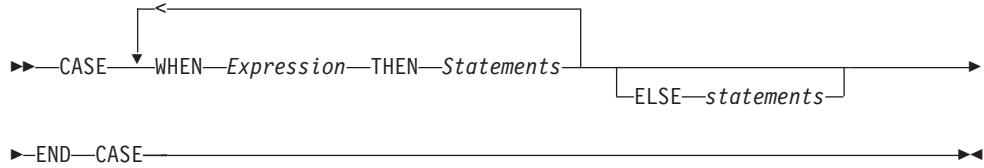
The CASE statement uses rules defined in WHEN clauses to select a block of statements to process.

There are two forms of the CASE statement: the simple form and the searched form.

Syntax



Searched CASE statement



In the simple form, the main expression is evaluated first. Each WHEN clause expression is evaluated in turn until the result is equal to the main expression's result. That WHEN clause's statements are then executed. If no match is found and the optional ELSE clause is present, the ELSE clause's statements are executed instead. The test values do not have to be literals. The only requirement is that the main expression and the WHEN clause expressions evaluate to types that can be compared.

In the searched form, each WHEN clause expression is evaluated in turn until one evaluates to TRUE. That WHEN clause's statements are then executed. If none of the expressions evaluates to TRUE and the optional ELSE clause is present, the ELSE clause's statements are executed. There does not have to be any similarity between the expressions in each CASE clause. The only requirement is that they all evaluate to a boolean value.

The ESQL language has both a CASE statement and a CASE function (see "CASE function" on page 939 for details of the CASE function). The CASE statement chooses one of a set of statements to execute. The CASE function chooses one of a set of expressions to evaluate and returns as its value the return value of the chosen expression.

Examples

Simple CASE statement:

```
CASE size
  WHEN minimum + 0 THEN
    SET description = 'small';
  WHEN minimum + 1 THEN
    SET description = 'medium';
  WHEN minimum + 2 THEN
    SET description = 'large';
    CALL handleLargeObject();
  ELSE
    SET description = 'unknown';
    CALL handleError();
END CASE;
```

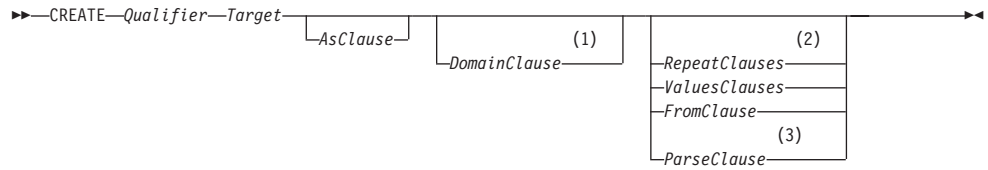
Searched CASE statement:

```
CASE
  WHEN i <> 0 THEN
    CALL handleI(i);
  WHEN j > 1 THEN
    CALL handleIZeroAndPositiveJ(j);
  ELSE
    CALL handleAllOtherCases(j);
END CASE;
```

CREATE statement

The CREATE statement creates a new message field.

Syntax



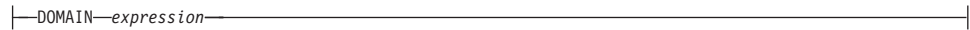
Qualifier:



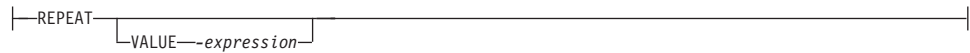
AsClause:



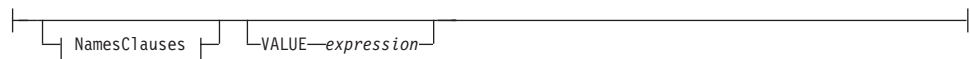
DomainClause:



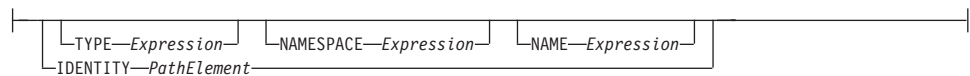
RepeatClauses:



Values clauses:



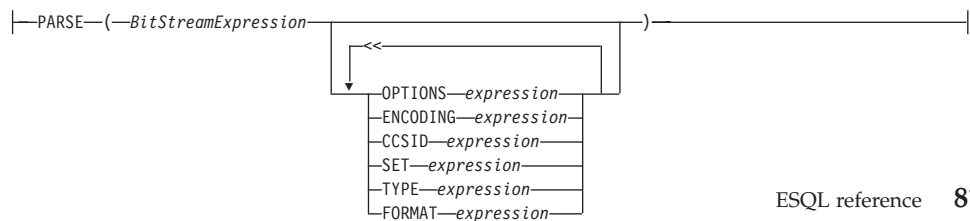
NamesClauses:



FromClause:



Parse clause:



Notes:

- 1 Do not use the *DomainClause* and *ParseClause* with the **FIELD** qualifier.
- 2 Use the *RepeatClause* only with the **PREVIOUSIBLING** and **NEXTSIBLING** qualifiers
- 3 Each subclause within the *ParseClause* can occur once only.

The new message field is positioned either at a given location (**CREATE FIELD**) or relative to a currently-existing location (**CREATE ... OF...**). New fields can be created only when the target field reference points to a modifiable message, for example `Environment`, `InputLocalEnvironment`, `OutputLocalEnvironment`, `OutputRoot`, or `OutputExceptionList`.

If you include a **FIELD** clause, the field specified by target is navigated to (creating the fields if necessary) and any values clause or from clause is executed. This form of **CREATE** statement does not necessarily create any fields at all; it ensures only that the given fields do exist.

If you use array indices in the target field reference, only one instance of a particular field can be created. Thus, if you write a **SET** statement starting:

```
SET OutputRoot.XML.Message.Structure[2].Field = ...
```

at least one instance of `Structure` must already exist in the message. That is, the only fields in the tree that are created are ones on a direct path from the root to the field identified by the field reference.

If you include a **PREVIOUSIBLING**, **NEXTSIBLING**, **FIRSTCHILD**, or **LASTCHILD** clause, the field specified by target is navigated to (creating the fields if necessary) in exactly the same way as for the **FIELD** clause. A new field is then created and attached in the specified position (for example as **PREVIOUSIBLING** or **FIRSTCHILD**). This form of **CREATE** statement always creates a new field and places it in the specified position.

If you use two **CREATE FIRSTCHILD OF** target statements that specify the same target, the second statement creates a new field as the first child of the target, and displaces the previously-created first child to the right in the message tree (so it is no longer the first child). Similarly, **CREATE LASTCHILD OF** target navigates to the target field and adds a new field as its rightmost child, displacing the previous last child to the left.

CREATE PREVIOUSIBLING OF target creates a field to the immediate left of the field specified by target (so the depth of the tree is not changed); similarly, **CREATE NEXTSIBLING OF** target creates a field to the immediate right of the field specified by target. When creating **PREVIOUSIBLING** or **NEXTSIBLING**, you can use the **REPEAT** keyword to copy the type and name of the new field from the current field.

AS clause:

If present, the **AS** clause moves the named reference variable to point at the newly-created field. This is useful because you probably want to involve the new field in some further processing.

DOMAIN clause:

If present, the DOMAIN clause associates the new field with a new parser of the specified type. This clause expects a root field name (for example, XML or MQRFH2). If the DOMAIN clause is present, but the value supplied is a zero-length character string, a new parser of the same type as the parser that owns the field specified by target is created. An exception is thrown if the supplied domain name is not CHARACTER data type or its value is NULL. Do not specify the DOMAIN clause with the FIELD clause; it is not certain that a new field is created.

REPEAT clause:

Use the REPEAT clause to copy the new field's type and name from the target field. Alternatively, the new field's type, name, and value can be:

- Copied from any existing field (using the FROM clause)
- Specified explicitly (using the VALUES clause)
- Defined by parsing a bit stream (using the PARSE clause)

In the case of the FROM and PARSE clauses, you can also create children of the new field.

VALUES clause:

For the VALUES clause, the type, name, and value (or any subset of these) can be specified by any expression that returns a suitable data type (INTEGER for type, CHARACTER for name, and any scalar type for value). An exception is thrown if the value supplied for a type or name is NULL.

NAMES clause:

The NAMES clause takes any expression that returns a non-null value of type character. The meaning depends on the presence of NAME and NAMESPACE clauses as follows:

NAMESPACE	NAME	Element named as follows
No	No	The element is nameless (name flag not automatically set)
No	Yes	The element is given the name in the default namespace
Yes	No	The element is given the empty name in the given namespace
Yes	Yes	The element is given the given name in the given namespace

The IDENTITY operand takes a single path element in place of the TYPE and NAME clauses, where a path element contains (at most) a type, a namespace, a name, and an index. These specify the type, namespace, name, and index of the element to be created and follow all the rules described in the topic for field references (see "ESQL field references" on page 792). For example:

```
IDENTITY (XML.attribute)Space1:Name1[42]
```

See the Examples section below for how to use the IDENTITY operand.

FROM clause:

For the FROM clause, the new field's type, name, and value are taken from the field pointed to by *SourceFieldReference*. Any existing child fields of the target are detached (the field could already exist in the case of a FIELD clause), and the new field is given copies of the source field's children, grandchildren, and so on.

PARSE clause:

If a PARSE clause is present, a subtree is built under the newly-created field from the supplied bit stream. The algorithm for doing this varies from parser to parser and according to the options specified. All parsers support the mode *RootBitStream*, in which the tree creation algorithm is the same as that used by an input node.

Some parsers also support a second mode, *FolderBitStream*, which generates a sub-tree from a bit stream created by the *ASBITSTREAM* function (see "ASBITSTREAM function" on page 925) using that mode.

When the statement is processed, any PARSE clause expressions are evaluated. An exception is thrown if any of the following expressions do not result in a non-null value of the appropriate type:

Clause	Type	Default value
Options	integer	RootBitStream & ValidateNone
Encoding	integer	0
Ccsid	integer	0
Message set	character	Zero length string
Message type	character	Zero length string
Message format	character	Zero length string

Although the OPTIONS clause accepts any expression that returns a value of type integer, it is only meaningful to generate option values from the list of supplied constants, using the BITOR function if more than one option is required.

Once generated, the value becomes an integer and you can save it in a variable or pass it as a parameter to a function, as well as using it directly with a CREATE statement. The list of globally defined constants is:

```
Validate master options...
ValidateContentAndValue
ValidateValue -- Can be used with ValidateContent
ValidateContent -- Can be used with ValidateValue
ValidateNone

Validate failure action options...
ValidateException
ValidateExceptionList
ValidateLocalError
ValidateUserTrace

Validate value constraints options...
ValidateFullConstraints
ValidateBasicConstraints

Validate fix up options...
ValidateFullFixUp
```

ValidateNoFixUp

Parse timing options...
ParseComplete
ParseImmediate
ParseOnDemand

Notes:

1. The `validateFullFixUp` option is reserved for future use. Selecting `validateFullFixUp` gives identical behavior to `validateNoFixUp`.
2. The `validateFullConstraints` option is reserved for future use. Selecting `validateFullConstraints` gives identical behavior to `validateBasicConstraints`.
3. For full details of the validation options, refer to “Validation properties for messages in the MRM domain” on page 703.
4. The *Validate timing options* correspond to *Parse Timing options* and, in particular, *Validate Deferred* is called *On Demand*.

You can specify only one option from each group, with the exception of `ValidateValue` and `ValidateContent`, which you can use together to obtain the content and value validation. If you do not specify an option within a group, the option in bold is used.

The `ENCODING` clause accepts any expression that returns a value of type integer. However, it is only meaningful to generate option values from the list of supplied constants:

```
MQENC_INTEGER_NORMAL  
MQENC_INTEGER_REVERSED  
MQENC_DECIMAL_NORMAL  
MQENC_DECIMAL_REVERSED  
MQENC_FLOAT_IEEE_NORMAL  
MQENC_FLOAT_IEEE_REVERSED  
MQENC_FLOAT_S390
```

The values used for the `CCSID` clause follow the normal numbering system. For example, 1200 = UCS-2, 1208 = UTF-8.

For absent clauses, the given default values are used. You are recommended to use the `CCSID` and encoding default values because these take their values from the queue manager’s encoding and `CCSID` settings.

Similarly, using the default values for each of the message set, type, and format options is useful, because many parsers do not require message set, type, or format information, so any valid value is sufficient.

When any expressions have been evaluated, a bit stream is parsed using the results of the expressions.

Note: Because this function has a large number of clauses, an alternative syntax is supported in which the parameters are supplied as a comma-separated list rather than by named clauses. In this case the expressions must be in the order:

```
ENCODING -> CCSID -> SET -> TYPE -> FORMAT -> OPTIONS
```

The list can be truncated at any point and an entirely empty expression can be used for any clauses where you do not supply a value.

Examples of using the CREATE statement

1. The following example creates the specified field:

```
CREATE FIELD OutputRoot.XML.Data;
```

2. The following example creates a field with no name, type, or value as the first child of ref1:

```
CREATE FIRSTCHILD OF ref1;
```

3. The following example creates a field using the specified type, name, and value:

```
CREATE NEXTSIBLING OF ref1 TYPE NameValue NAME 'Price' VALUE 92.3;
```

4. The following example creates a field with a type and name, but no value; the field is added before the sibling indicated by the dynamic reference (ref1):

```
CREATE PREVIOUSIBLING OF ref1 TYPE Name NAME 'Quantity';
```

5. The following example creates a field named Component, and moves the reference variable targetCursor to point at it:

```
CREATE FIRSTCHILD OF targetCursor AS targetCursor NAME 'Component';
```

The following example creates a new field as the right sibling of the field pointed to by the reference variable targetCursor having the same type and name as that field. The statement then moves targetCursor to point at the new field:

```
CREATE NEXTSIBLING OF targetCursor AS targetCursor REPEAT;
```

6. The following example shows how to use the PARSE clause:

```
DECLARE bodyBlob BLOB ASBITSTREAM(InputRoot.XML, InputProperties.Encoding,  
    InputProperties.CodedCharSetId);  
DECLARE creationPtr REFERENCE TO OutputRoot;  
CREATE LASTCHILD OF creationPtr DOMAIN('XML') PARSE(bodyBlob,  
    InputProperties.Encoding,  
    InputProperties.CodedCharSetId);
```

This example can be extended to show the serializing and parsing of a field or folder:

```
DECLARE bodyBlob BLOB ASBITSTREAM(InputRoot.XML.TestCase.myFolder,  
    InputProperties.Encoding,  
    InputProperties.CodedCharSetId,"","FolderBitStream");  
DECLARE creationPtr REFERENCE TO OutputRoot;  
CREATE LASTCHILD OF creationPtr DOMAIN('XML') PARSE(bodyBlob,  
    InputProperties.Encoding,  
    InputProperties.CodedCharSetId,"","FolderBitStream");
```

7. The following example shows how to use the IDENTITY operand:

```
| CREATE FIELD OutputRoot.XMLNS.TestCase.Root IDENTITY (XML.ParserRoot)Root;  
| CREATE FIELD OutputRoot.XMLNS.TestCase.Root.Attribute  
|     IDENTITY (XML.Attribute)NSpace1:Attribute VALUE 'Attrib Value';  
| CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Root  
|     IDENTITY (XML.Element)NSpace1:Element1[1] VALUE 'Element 1 Value';  
| CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Root  
|     IDENTITY (XML.Element)NSpace1:Element1[2] VALUE 'Element 2 Value';  
|  
|  
|  
|
```

This sequence of statements produces the following output message:

```
| <TestCase>  
|   <Root xmlns:NS1="NSpace1" NS1:Attribute="Attrib Value">  
|     <NS1:Element1>Element 1 Value</NS1:Element1>  
|     <NS1:Element1>Element 2 Value</NS1:Element1>  
|   </Root>  
| </TestCase>
```

8. The following example shows how you can use the DOMAIN clause to avoid losing information unique to the XML parser when an unlike parser copy occurs:


```

DECLARE bodyBlob BLOB ASBITSTREAM(InputRoot.XML, InputProperties.Encoding,
InputProperties.CodedCharSetId);
CREATE FIELD Environment.Variables.myXMLTree;
DECLARE creationPtr REFERENCE TO Environment.Variables.myXMLTree;
CREATE FIRSTCHILD OF creationPtr DOMAIN('XML') PARSE(bodyBlob,
InputProperties.Encoding,
InputProperties.CodedCharSetId);

```

Example of CREATE statement

This example provides sample ESQL and an input message, which produce the output message at the end of the example.

```

CREATE COMPUTE MODULE CreateStatement_Compute
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    CALL CopyMessageHeaders();

    CREATE FIELD OutputRoot.XML.TestCase.description TYPE NameValue VALUE 'This is my TestCase' ;
    DECLARE cursor REFERENCE TO OutputRoot.XML.TestCase;
    CREATE FIRSTCHILD OF cursor Domain('XML')
        NAME 'Identifier' VALUE InputRoot.XML.TestCase.Identifier;
    CREATE LASTCHILD OF cursor Domain('XML') NAME 'Sport' VALUE InputRoot.XML.TestCase.Sport;
    CREATE LASTCHILD OF cursor Domain('XML') NAME 'Date' VALUE InputRoot.XML.TestCase.Date;
    CREATE LASTCHILD OF cursor Domain('XML') NAME 'Type' VALUE InputRoot.XML.TestCase.Type;
    CREATE FIELD cursor.Division[1].Number TYPE NameValue VALUE 'Premiership';
    CREATE FIELD cursor.Division[1].Result[1].Number TYPE NameValue VALUE '1' ;
    CREATE FIELD cursor.Division[1].Result[1].Home TYPE Name;
    CREATE LASTCHILD OF cursor.Division[1].Result[1].Home NAME 'Team' VALUE 'Liverpool' ;
    CREATE LASTCHILD OF cursor.Division[1].Result[1].Home NAME 'Score' VALUE '4';
    CREATE FIELD cursor.Division[1].Result[1].Away TYPE Name;
    CREATE LASTCHILD OF cursor.Division[1].Result[1].Away NAME 'Team' VALUE 'Everton';
    CREATE LASTCHILD OF cursor.Division[1].Result[1].Away NAME 'Score' VALUE '0';
    CREATE FIELD cursor.Division[1].Result[2].Number TYPE NameValue VALUE '2';
    CREATE FIELD cursor.Division[1].Result[2].Home TYPE Name;
    CREATE LASTCHILD OF cursor.Division[1].Result[2].Home NAME 'Team' VALUE 'Manchester United';
    CREATE LASTCHILD OF cursor.Division[1].Result[2].Home NAME 'Score' VALUE '2';
    CREATE FIELD cursor.Division[1].Result[2].Away TYPE Name;
    CREATE LASTCHILD OF cursor.Division[1].Result[2].Away NAME 'Team' VALUE 'Arsenal';
    CREATE LASTCHILD OF cursor.Division[1].Result[2].Away NAME 'Score' VALUE '3';
    CREATE FIELD cursor.Division[2].Number TYPE NameValue VALUE '2';
    CREATE FIELD cursor.Division[2].Result[1].Number TYPE NameValue VALUE '1';
    CREATE FIELD cursor.Division[2].Result[1].Home TYPE Name;
    CREATE LASTCHILD OF cursor.Division[2].Result[1].Home NAME 'Team' VALUE 'Port Vale';
    CREATE LASTCHILD OF cursor.Division[2].Result[1].Home NAME 'Score' VALUE '9' ;
    CREATE FIELD cursor.Division[2].Result[1].Away TYPE Name;
    CREATE LASTCHILD OF cursor.Division[2].Result[1].Away NAME 'Team' VALUE 'Brentford';
    CREATE LASTCHILD OF cursor.Division[2].Result[1].Away NAME 'Score' VALUE '5';

END;

CREATE PROCEDURE CopyMessageHeaders() BEGIN
    DECLARE I INTEGER 1;
    DECLARE J INTEGER CARDINALITY(InputRoot.*[I]);
    WHILE I < J DO
        SET OutputRoot.*[I] = InputRoot.*[I];
        SET I = I + 1;
    END WHILE;
END;

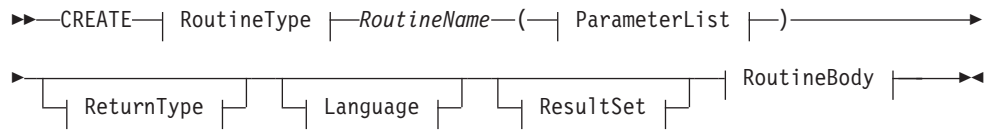
END MODULE;

```

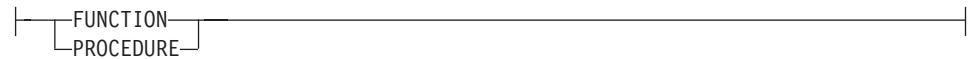
CREATE FUNCTION statement

The CREATE FUNCTION and CREATE PROCEDURE statements define a callable function or procedure, usually called a routine.

Syntax



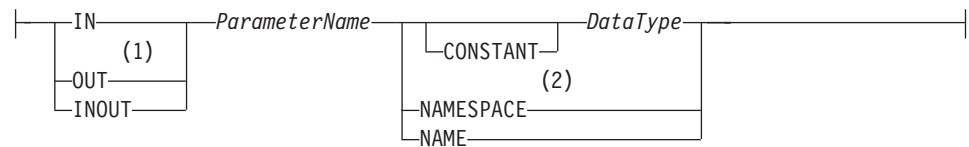
RoutineType:



ParameterList:



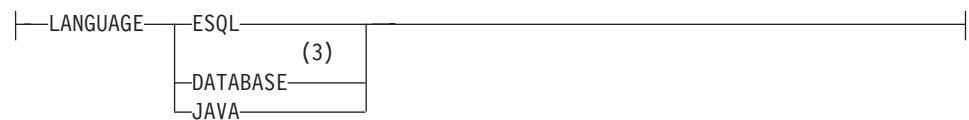
Parameter:



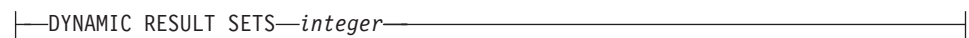
ReturnType:



Language:



ResultSet:



RoutineBody:



Notes:

1. If the routine type is FUNCTION, the direction indicator (IN, OUT, INOUT) is optional for each parameter. However, it is recommended that

you specify a direction indicator for all new routines of any type.

2. When the `NAMESPACE` or `NAME` clause is used, its value is implicitly constant and of type `CHARACTER`. For information on the use of `CONSTANT` variables, see the “`DECLARE` statement” on page 851.
3. If the routine type is `FUNCTION`, you cannot specify a `LANGUAGE` of `DATABASE`.

Overview

The `CREATE FUNCTION` and `CREATE PROCEDURE` statements define a callable function or procedure, usually called a routine.

Note: In previous versions of the product, `CREATE FUNCTION` and `CREATE PROCEDURE` had different uses and different capabilities. However, they have since been enhanced to the point where only a few differences remain. The only ways in which functions differ from procedures are listed in notes 1 and 3 below the syntax diagram.

Routines are useful for creating reusable blocks of code that can be executed independently many times. They can be implemented as a series of `ESQL` statements, a Java method, or a database stored procedure. This flexibility means that some of the clauses in the syntax diagram are not applicable (or allowed) for all types of routine.

Each routine has a name, which must be unique within the schema to which it belongs. This means that routine names cannot be overloaded; if the broker detects that a routine has been overloaded, it raises an exception.

The `LANGUAGE` clause specifies the language in which the routine’s body is written. The options are:

DATABASE

The procedure is called as a database stored procedure.

ESQL

The procedure is called as an `ESQL` routine.

JAVA

The procedure is called as a static method in a Java class.

Unspecified

If you do not specify the `LANGUAGE` clause the default language is `ESQL`, unless you specify the `EXTERNAL NAME` clause, in which case the default language is `DATABASE`.

There are restrictions on the use of the `LANGUAGE` clause. Do not use:

- The `ESQL` option with an `EXTERNAL NAME` clause
- The `DATABASE` or `JAVA` options without an `EXTERNAL NAME` clause
- The `DATABASE` option with a routine type of `FUNCTION`

Specify the routine’s name using the *RoutineName* clause and the routine’s parameters using the *ParameterList* clause. If the `LANGUAGE` clause specifies `ESQL`, the routine must be implemented using a single `ESQL` statement. This statement is most useful if it is a compound statement (`BEGIN ... END`) as it can then contain as many `ESQL` statements as necessary to fulfil its function.

Alternatively, instead of providing an ESQL body for the routine, you can specify a LANGUAGE clause other than ESQL. This allows you to use the EXTERNAL NAME clause to provide a reference to the actual body of the routine, wherever it is located externally to the broker. For more details about using the EXTERNAL NAME clause, see “Invoking stored procedures” on page 212 and Calling a Java routine.

Routines of any LANGUAGE type can have IN, OUT, and INOUT parameters. This allows the caller to pass several values into the routine, and to receive several updated values back. This is in addition to any RETURNS clause the routine may have. The RETURNS clause allows the routine to pass back a value to the caller.

Routines implemented in different languages have their own restrictions on which data-types can be passed in or returned, and these are documented below. The data-type of the returned value must match the data-type of the value defined to be returned from the routine. Also, if a routine is defined to have a return value, the caller of the routine cannot ignore it. For more details see the “CALL statement” on page 813.

Routines can be defined in either a module or a schema. Routines defined in a module are local in scope to the current node, which means that only code belonging to that same module (or node) can invoke them. Routines defined in schema scope, however, can be invoked by either of the following:

- Code in the same schema.
- Code in any other schema, if either of the following applies:
 1. The other schema’s PATH clause contains the path to the called routine, *or*
 2. The called routine is invoked using its fully-qualified name (which is its name, prefixed by its schema name, separated by a period).

Thus, if you need to invoke the same routine in more than one node, define it in a schema.

For any language or routine type, the method of invocation of the routine must match the manner of declaration of the routine. If the routine has a RETURNS clause, use either the FUNCTION invocation syntax or a CALL statement with an INTO clause. Conversely, if a routine has no RETURNS clause you must use a CALL statement without an INTO clause.

Parameter directions

Parameters passed to routines always have a direction associated with them. This can be any one of:

IN The value of the parameter cannot be changed by the routine. A NULL value for the parameter is allowed and can be passed to the routine.

OUT

When it is received by the called routine, the parameter passed into the routine always has a NULL value of the correct data type. This happens irrespective of its value before the routine is called. The routine is allowed to change the value of the parameter.

INOUT

INOUT is both an IN and an OUT parameter. It passes a value into the routine, and the value passed in can be changed by the routine. A NULL value for the parameter is allowed and can be passed both into and out from the routine.

If the routine type is FUNCTION, the direction indicator (IN, OUT, INOUT) is optional for each parameter. However, it is strongly recommended that you specify a direction indicator for all new routines of any type.

ESQL variables that are declared to be CONSTANT (or references to variables declared to be CONSTANT) are not allowed to have the direction OUT or INOUT.

ESQL routines

ESQL routines are written in ESQL, and have a LANGUAGE clause of ESQL. The body of an ESQL routine is usually a compound statement of the form BEGIN ... END, containing multiple statements for processing the parameters passed to the routine.

ESQL example 1

The following example shows the same procedure as in “Database routine example 1” on page 847, but implemented as an ESQL routine rather than as a stored procedure. The CALL syntax and results of this routine are the same as those in “Restrictions on Java routines” on page 834.

```
CREATE PROCEDURE swapParms (
  IN parm1 CHARACTER,
  OUT parm2 CHARACTER,
  INOUT parm3 CHARACTER )
BEGIN
  SET parm2 = parm3;
  SET parm3 = parm1;
END;
```

ESQL example 2

This example procedure shows the recursive use of an ESQL routine. It parses a tree, visiting all places at and below the specified starting point, and reports what it has found:

```
SET OutputRoot.MQMD = InputRoot.MQMD;

DECLARE answer CHARACTER;
SET answer = '';

CALL navigate(InputRoot.XML, answer);
SET OutputRoot.XML.Data.FieldNames = answer;

CREATE PROCEDURE navigate (IN root REFERENCE, INOUT answer CHARACTER)
BEGIN
  SET answer = answer || 'Reached Field... Type:'
  || CAST(FIELDTYPE(root) AS CHAR)||
  ': Name:' || FIELDNAME(root) || ': Value : ' || root || ': ';

  DECLARE cursor REFERENCE TO root;
  MOVE cursor FIRSTCHILD;
  IF LASTMOVE(cursor) THEN
    SET answer = answer || 'Field has children... drilling down ';
  ELSE
    SET answer = answer || 'Listing siblings... ';
  END IF;

  WHILE LASTMOVE(cursor) DO
    CALL navigate(cursor, answer);
    MOVE cursor NEXTSIBLING;
```

```

END WHILE;

SET answer = answer || 'Finished siblings... Popping up ';
END;

```

When given the following input message:

```

<Person>
  <Name>John Smith</Name>
  <Salary period='monthly' taxable='yes'>-1200</Salary>
</Person>

```

the procedure produces the following output, which has been manually formatted:

```

Reached Field... Type:16777232: Name:XML: Value :: Field has children...
drilling down
Reached Field... Type:16777216: Name:Person: Value :: Field has children...
drilling down
Reached Field... Type:16777216: Name:Name:
Value :John Smith: Field has children... drilling down
Reached Field... Type:33554432: Name::
Value :John Smith: Listing siblings... Finished siblings... Popping up
Finished siblings... Popping up
Reached Field... Type:16777216: Name:Salary:
Value :-1200: Field has children... drilling down
Reached Field... Type:50331648: Name:period:
Value :monthly: Listing siblings... Finished siblings... Popping up
Reached Field... Type:50331648: Name:taxable:
Value :yes: Listing siblings... Finished siblings... Popping up
Reached Field... Type:33554432: Name::
Value :-1200: Listing siblings... Finished siblings... Popping up
Finished siblings... Popping up
Finished siblings... Popping up
Finished siblings... Popping up

```

Java routines

A Java routine is implemented as a Java method and has a LANGUAGE clause of JAVA. For Java routines, the *ExternalRoutineName* must contain the class name and method name of the Java method to be called. Specify the *ExternalRoutineName* like this:

```
>>--"-- className---.---methodName--"-----<<
```

where *className* identifies the class that contains the method and *methodName* identifies the method to be invoked. If the class is part of a package, the class identifier part must include the complete package prefix; for example, "com.ibm.broker.test.MyClass.myMethod".

To find the Java class, the broker searches as described in "Deploying Java classes" on page 834.

Any Java method you want to invoke must have the following basic signature:

```
public static <return-type> <method-name> (< 0 - N parameters>)
```

where <return-type> must be in the list of Java IN data types in the table in "ESQL to Java data type mapping" on page 833 (excluding the REFERENCE type, which is not permitted as a return value), or the Java void data type. The parameter data types must also be in the "ESQL to Java data type mapping" on page 833 table. In addition, the Java method is not allowed to have an exception throws clause in its signature.

The Java method's signature must match the ESQL routine's declaration of the method. Also, you must observe the following rules:

- Ensure that the Java method name, including the class name and any package qualifiers, matches the procedure's EXTERNAL NAME.
- If the Java return type is void, do not put a RETURNS clause on the ESQL routine's definition. Conversely, if the Java return type is *not void*, you must put a RETURNS clause on the ESQL routine's definition.
- Ensure that every parameter's type and direction matches the ESQL declaration, according to the rules listed in the table in "ESQL to Java data type mapping" on page 833.
- Ensure that the method's return type matches the data type of the RETURNS clause.
- Enclose EXTERNAL NAME in quotation marks because it must contain at least "class.method".
- If you want to invoke an overloaded Java method, you must create a separate ESQL definition for each overloaded method and give each ESQL definition a unique routine name.

You can use the Java User defined Node (UDN) API in your Java method, provided that you observe the restrictions documented in "Restrictions on Java routines" on page 834. For more information about using the UDN API, see Compiling a Java user-defined node .

Java routine example 1

This routine contains three parameters of varying directions, and returns an integer, which maps to a Java return type of `java.lang.Long`.

```
CREATE FUNCTION myProc1( IN P1 INTEGER, OUT P2 INTEGER, INOUT P3 INTEGER )
  RETURNS INTEGER
  LANGUAGE JAVA
  EXTERNAL NAME "com.ibm.broker.test.MyClass.myMethod1";
```

You can use the following ESQL to invoke `myProc1`:

```
CALL myProc1( intVar1, intVar2, intVar3) INTO intReturnVar3;
-- or
SET intReturnVar3 = myProc1( intVar1, intVar2, intVar3);
```

Java routine example 2

This routine contains three parameters of varying directions and has a Java return type of `void`.

```
CREATE PROCEDURE myProc2( IN P1 INTEGER, OUT P2 INTEGER, INOUT P3 INTEGER )
  LANGUAGE JAVA
  EXTERNAL NAME "com.ibm.broker.test.MyClass.myMethod2";
```

You must use the following ESQL to invoke `myProc2`:

```
CALL myProc2(intVar1, intVar2, intVar3);
```

The following Java class provides a method for each of the preceding Java examples:

```
package com.ibm.broker.test;

class MyClass {
  public static Long myMethod1( Long P1, Long[] P2 Long[] P3) { ... }
  public static void myMethod2( Long P2, Long[] P2 Long[] P3) { ... }

  /* When either of these methods is called:
     P1 may or may not be NULL (depending on the value of intVar1).
```


P2[0] is always NULL (whatever the value of intVar2).
P3[0] may or may not be NULL (depending on the value of intVar3).
This is the same as with LANGUAGE ESQL routines.
When these methods return:
 intVar1 is unchanged
 intVar2 may still be NULL or may have been changed
 intVar3 may contain the same value or may have been changed.
This is the same as with LANGUAGE ESQL routines.

When myMethod1 returns: intReturnVar3 is either NULL (if the method returns NULL) or it contains the value returned by the method.

```
*/
}
```

ESQL to Java data type mapping

The following table summarizes the mappings from ESQL to Java.

Notes:

- Only the Java scalar wrappers are passed to Java.
- The ESQL scalar types are mapped to Java data types as object wrappers, or object wrapper arrays, depending upon the direction of the procedure parameter. Each wrapper array contains exactly one element.
- Scalar object wrappers are used to allow NULL values to be passed to and from Java methods.

ESQL data types ¹	Java IN data types	Java INOUT and OUT data types
INTEGER, INT	java.lang.Long	java.lang.Long []
FLOAT	java.lang.Double	java.lang.Double[]
DECIMAL	java.math.BigDecimal	java.math.BigDecimal[]
CHARACTER, CHAR	java.lang.String	java.lang.String[]
BLOB	byte[]	byte[][]
BIT	java.util.BitSet	java.util.BitSet[]
DATE	com.ibm.broker.plugin.MbDate	com.ibm.broker.plugin.MbDate[]
TIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
GMTTIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
TIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
GMTTIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
INTERVAL	Not supported	Not supported
BOOLEAN	java.lang.Boolean	java.lang.Boolean[]
REFERENCE (to a message tree) ^{3 4} _{5 6}	com.ibm.broker.plugin.MbElement	com.ibm.broker.plugin.MbElement[] (Supported for INOUT. Not supported for OUT)
ROW	Not supported	Not supported
LIST	Not supported	Not supported

1. Variables that are declared to be CONSTANT (or references to variables declared to be CONSTANT) are not allowed to have the direction INOUT or OUT.
2. The time zone set in the Java variable is not important; you obtain the required time zone in the output ESQL.

3. The reference parameter cannot be NULL when passed into a Java method.
4. The reference cannot have the direction OUT when passed into a Java method.
5. If an *MbElement* is passed back from Java to ESQL as an INOUT parameter, it must point to a location in the same message tree as that pointed to by the *MbElement* that was passed into the called Java method.

For example, if an ESQL reference to `OutputRoot.XML.Test` is passed into a Java method as an INOUT *MbElement*, but a different *MbElement* is passed back to ESQL when the call returns, the different element must also point to somewhere in the `OutputRoot` tree.

6. An *MbElement* cannot be returned from a Java method with the RETURNS clause, because no ESQL routine can return a reference. However, an *MbElement* can be returned as an INOUT direction parameter, subject to the conditions described in point 5 above.

A REFERENCE to a scalar variable can be used in the CALL of a Java method, provided that the data type of the variable the reference refers to matches the corresponding data type in the Java program signature.

Restrictions on Java routines

The following restrictions apply to Java routines called from ESQL:

- You must ensure that the Java method is threadsafe (reentrant).
- The only database connections permitted are JDBC type 4 connections. Furthermore, database operations are not part of a broker transaction; this means that they cannot be controlled by an external resource coordinator (such as would be the case in an XA environment).
- The Java User defined Node (UDN) API should be used only by the same thread that invoked the Java method.

You are allowed to spawn threads inside your method. However, spawned threads must not use the Java plug-in APIs and you must return control back to the broker.

Note that all restrictions that apply to the usage of the UDN API also apply to Java methods called from ESQL.

- Java methods called from ESQL must not use the *MbNode* class. This means that they cannot create objects of type *MbNode*, or call any of the methods on an existing *MbNode* object.
- If you want to perform MQ or JMS work inside a Java method called from ESQL, you must follow the User Defined Node (UDN) guidelines for performing MQ and JMS work in a UDN.

Deploying Java classes

We recommend that you deploy your Java classes inside a Java Archive (JAR) file. There are two ways to deploy a JAR file to the broker:

1. By adding it to the Broker Archive (BAR) file

This is the recommended method.

You can add a JAR file to the BAR file manually, by hand, or automatically, using the tooling. It is recommended that you use the tooling.

If the tooling finds the correct Java class inside a referenced Java project open in the workspace, it automatically compiles the Java class into a JAR file and

adds it to the BAR file. This is the same procedure that you follow to deploy a Java Compute node inside a JAR, as described in User-defined node classloading.

When deploying a JAR file from the tooling, a redeploy of the BAR file containing the JAR file causes the referenced Java classes to be reloaded by the flow that has been redeployed; as does stopping and restarting a message flow that references a Java class. Ensure that you stop and restart (or redeploy) all flows that reference the JAR file that you want to update. This avoids the problem of some flows running with the old version of the JAR file and other flows running with the new version.

Note that the tooling will only deploy a JAR file; it will not deploy a standalone Java class file.

2. **By placing it in either of the following:**

- a. The <Workpath>/shared-classes/ folder on the machine running the broker
- b. The CLASSPATH environment variable on the machine running the broker

This procedure must be done manually; you cannot use the tooling.

In this method, redeploying the message flow does not reload the referenced Java classes; neither does stopping and restarting the message flow. The only way to reload the classes in this case is to stop and restart the broker itself.

Note that although you can deploy a standalone Java class by placing it in one of the two locations above, it is still recommended that you use a JAR file instead.

To enable the broker to find a Java class, ensure that it is in one of the above locations. If the broker cannot find the specified class, it throws an exception.

Although you have the choices shown above when deploying the JAR file, it is recommended that you choose the first method (allowing the tooling to deploy the BAR file) because this provides the greatest flexibility when redeploying the JAR file.

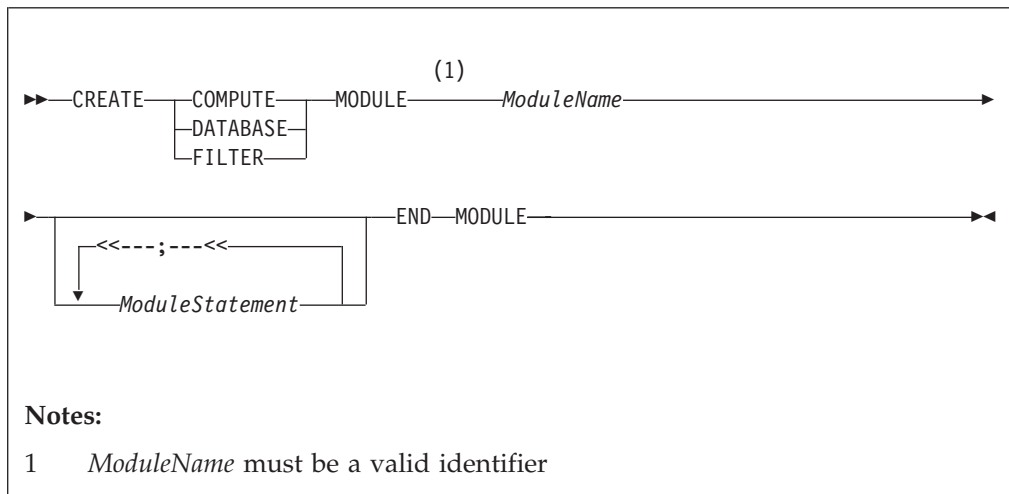
Database routines

CREATE FUNCTION does not support database routines. Use CREATE PROCEDURE to define a database routine.

CREATE MODULE statement

The CREATE MODULE statement creates a module, which is a named container associated with a node.

Syntax



A module in the Eclipse tools is referred to from a message processing node by name. The module must be in the <node schema>.

Module names occupy the same symbol space as functions and procedures defined in the schema. That is, modules, functions, and procedures contained by a schema must all have unique names.

Note: You are warned if there is no module associated with an ESQL node. You cannot deploy a flow containing a node in which a module is missing.

The modules for the Compute node, Database node, and Filter node must all contain exactly one function called Main. This function should return a Boolean. It is the entry point used by a message flow node when processing a message.

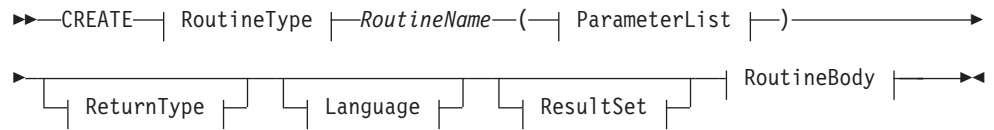
Correlation name	Compute module	Filter module	Database module
Database	×	×	×
Environment	×	×	×
Root		×	×
Body		×	×
Properties		×	×
ExceptionList		×	×
LocalEnvironment		×	×
InputRoot	×		
InputBody	×		
InputProperties	×		
InputExceptionList	×		
InputLocalEnvironment	×		
OutputRoot	×		
OutputExceptionList	×		
OutputLocalEnvironment	×		
DestinationList	Deprecated synonym for LocalEnvironment		

Correlation name	Compute module	Filter module	Database module
InputDestinationList	Deprecated synonym for InputLocalEnvironment		
OutputDestinationList	Deprecated synonym for OutputLocalEnvironment		

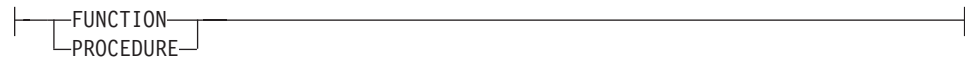
CREATE PROCEDURE statement

The CREATE FUNCTION and CREATE PROCEDURE statements define a callable function or procedure, usually called a routine.

Syntax



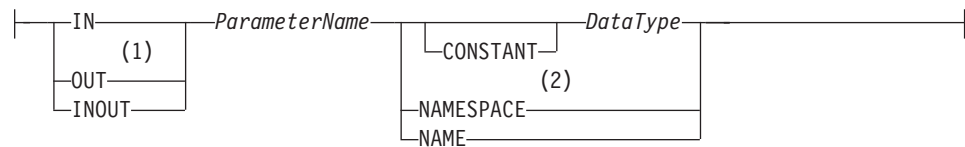
RoutineType:



ParameterList:



Parameter:



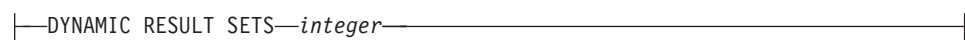
ReturnType:



Language:



ResultSet:



RoutineBody:



Notes:

1. If the routine type is FUNCTION, the direction indicator (IN, OUT, INOUT) is optional for each parameter. However, it is recommended that

you specify a direction indicator for all new routines of any type.

2. When the `NAMESPACE` or `NAME` clause is used, its value is implicitly constant and of type `CHARACTER`. For information on the use of `CONSTANT` variables, see the “`DECLARE` statement” on page 851.
3. If the routine type is `FUNCTION`, you cannot specify a `LANGUAGE` of `DATABASE`.

Overview

The `CREATE FUNCTION` and `CREATE PROCEDURE` statements define a callable function or procedure, usually called a routine.

Note: In previous versions of the product, `CREATE FUNCTION` and `CREATE PROCEDURE` had different uses and different capabilities. However, they have since been enhanced to the point where only a few differences remain. The only ways in which functions differ from procedures are listed in notes 1 and 3 below the syntax diagram.

Routines are useful for creating reusable blocks of code that can be executed independently many times. They can be implemented as a series of `ESQL` statements, a Java method, or a database stored procedure. This flexibility means that some of the clauses in the syntax diagram are not applicable (or allowed) for all types of routine.

Each routine has a name, which must be unique within the schema to which it belongs. This means that routine names cannot be overloaded; if the broker detects that a routine has been overloaded, it raises an exception.

The `LANGUAGE` clause specifies the language in which the routine’s body is written. The options are:

DATABASE

The procedure is called as a database stored procedure.

ESQL

The procedure is called as an `ESQL` routine.

JAVA

The procedure is called as a static method in a Java class.

Unspecified

If you do not specify the `LANGUAGE` clause the default language is `ESQL`, unless you specify the `EXTERNAL NAME` clause, in which case the default language is `DATABASE`.

There are restrictions on the use of the `LANGUAGE` clause. Do not use:

- The `ESQL` option with an `EXTERNAL NAME` clause
- The `DATABASE` or `JAVA` options without an `EXTERNAL NAME` clause
- The `DATABASE` option with a routine type of `FUNCTION`

Specify the routine’s name using the *RoutineName* clause and the routine’s parameters using the *ParameterList* clause. If the `LANGUAGE` clause specifies `ESQL`, the routine must be implemented using a single `ESQL` statement. This statement is most useful if it is a compound statement (`BEGIN ... END`) as it can then contain as many `ESQL` statements as necessary to fulfil its function.

Alternatively, instead of providing an ESQL body for the routine, you can specify a LANGUAGE clause other than ESQL. This allows you to use the EXTERNAL NAME clause to provide a reference to the actual body of the routine, wherever it is located externally to the broker. For more details about using the EXTERNAL NAME clause, see “Invoking stored procedures” on page 212 and Calling a Java routine.

Routines of any LANGUAGE type can have IN, OUT, and INOUT parameters. This allows the caller to pass several values into the routine, and to receive several updated values back. This is in addition to any RETURNS clause the routine may have. The RETURNS clause allows the routine to pass back a value to the caller.

Routines implemented in different languages have their own restrictions on which data-types can be passed in or returned, and these are documented below. The data-type of the returned value must match the data-type of the value defined to be returned from the routine. Also, if a routine is defined to have a return value, the caller of the routine cannot ignore it. For more details see the “CALL statement” on page 813.

Routines can be defined in either a module or a schema. Routines defined in a module are local in scope to the current node, which means that only code belonging to that same module (or node) can invoke them. Routines defined in schema scope, however, can be invoked by either of the following:

- Code in the same schema.
- Code in any other schema, if either of the following applies:
 1. The other schema’s PATH clause contains the path to the called routine, *or*
 2. The called routine is invoked using its fully-qualified name (which is its name, prefixed by its schema name, separated by a period).

Thus, if you need to invoke the same routine in more than one node, define it in a schema.

For any language or routine type, the method of invocation of the routine must match the manner of declaration of the routine. If the routine has a RETURNS clause, use either the FUNCTION invocation syntax or a CALL statement with an INTO clause. Conversely, if a routine has no RETURNS clause you must use a CALL statement without an INTO clause.

Parameter directions

Parameters passed to routines always have a direction associated with them. This can be any one of:

IN The value of the parameter cannot be changed by the routine. A NULL value for the parameter is allowed and can be passed to the routine.

OUT

When it is received by the called routine, the parameter passed into the routine always has a NULL value of the correct data type. This happens irrespective of its value before the routine is called. The routine is allowed to change the value of the parameter.

INOUT

INOUT is both an IN and an OUT parameter. It passes a value into the routine, and the value passed in can be changed by the routine. A NULL value for the parameter is allowed and can be passed both into and out from the routine.

If the routine type is FUNCTION, the direction indicator (IN, OUT, INOUT) is optional for each parameter. However, it is strongly recommended that you specify a direction indicator for all new routines of any type.

ESQL variables that are declared to be CONSTANT (or references to variables declared to be CONSTANT) are not allowed to have the direction OUT or INOUT.

ESQL routines

ESQL routines are written in ESQL, and have a LANGUAGE clause of ESQL. The body of an ESQL routine is usually a compound statement of the form BEGIN ... END, containing multiple statements for processing the parameters passed to the routine.

ESQL example 1

The following example shows the same procedure as in “Database routine example 1” on page 847, but implemented as an ESQL routine rather than as a stored procedure. The CALL syntax and results of this routine are the same as those in “Restrictions on Java routines” on page 834.

```
CREATE PROCEDURE swapParms (
  IN parm1 CHARACTER,
  OUT parm2 CHARACTER,
  INOUT parm3 CHARACTER )
BEGIN
  SET parm2 = parm3;
  SET parm3 = parm1;
END;
```

ESQL example 2

This example procedure shows the recursive use of an ESQL routine. It parses a tree, visiting all places at and below the specified starting point, and reports what it has found:

```
SET OutputRoot.MQMD = InputRoot.MQMD;

DECLARE answer CHARACTER;
SET answer = '';

CALL navigate(InputRoot.XML, answer);
SET OutputRoot.XML.Data.FieldNames = answer;

CREATE PROCEDURE navigate (IN root REFERENCE, INOUT answer CHARACTER)
BEGIN
  SET answer = answer || 'Reached Field... Type:'
  || CAST(FIELDTYPE(root) AS CHAR)||
  ': Name:' || FIELDNAME(root) || ': Value : ' || root || ': ';

  DECLARE cursor REFERENCE TO root;
  MOVE cursor FIRSTCHILD;
  IF LASTMOVE(cursor) THEN
    SET answer = answer || 'Field has children... drilling down ';
  ELSE
    SET answer = answer || 'Listing siblings... ';
  END IF;

  WHILE LASTMOVE(cursor) DO
    CALL navigate(cursor, answer);
    MOVE cursor NEXTSIBLING;
```

```

END WHILE;

SET answer = answer || 'Finished siblings... Popping up ';
END;

```

When given the following input message:

```

<Person>
  <Name>John Smith</Name>
  <Salary period='monthly' taxable='yes'>-1200</Salary>
</Person>

```

the procedure produces the following output, which has been manually formatted:

```

Reached Field... Type:16777232: Name:XML: Value :: Field has children...
drilling down
Reached Field... Type:16777216: Name:Person: Value :: Field has children...
drilling down
Reached Field... Type:16777216: Name:Name:
Value :John Smith: Field has children... drilling down
Reached Field... Type:33554432: Name::
Value :John Smith: Listing siblings... Finished siblings... Popping up
Finished siblings... Popping up
Reached Field... Type:16777216: Name:Salary:
Value :-1200: Field has children... drilling down
Reached Field... Type:50331648: Name:period:
Value :monthly: Listing siblings... Finished siblings... Popping up
Reached Field... Type:50331648: Name:taxable:
Value :yes: Listing siblings... Finished siblings... Popping up
Reached Field... Type:33554432: Name::
Value :-1200: Listing siblings... Finished siblings... Popping up
Finished siblings... Popping up
Finished siblings... Popping up
Finished siblings... Popping up

```

Java routines

A Java routine is implemented as a Java method and has a LANGUAGE clause of JAVA. For Java routines, the *ExternalRoutineName* must contain the class name and method name of the Java method to be called. Specify the *ExternalRoutineName* like this:

```
>>--"--- className---.---methodName--"-----<<
```

where *className* identifies the class that contains the method and *methodName* identifies the method to be invoked. If the class is part of a package, the class identifier part must include the complete package prefix; for example, "com.ibm.broker.test.MyClass.myMethod".

To find the Java class, the broker searches as described in "Deploying Java classes" on page 834.

Any Java method you want to invoke must have the following basic signature:

```
public static <return-type> <method-name> (< 0 - N parameters>)
```

where <return-type> must be in the list of Java IN data types in the table in "ESQL to Java data type mapping" on page 833 (excluding the REFERENCE type, which is not permitted as a return value), or the Java void data type. The parameter data types must also be in the "ESQL to Java data type mapping" on page 833 table. In addition, the Java method is not allowed to have an exception throws clause in its signature.

The Java method's signature must match the ESQL routine's declaration of the method. Also, you must observe the following rules:

- Ensure that the Java method name, including the class name and any package qualifiers, matches the procedure's EXTERNAL NAME.
- If the Java return type is void, do not put a RETURNS clause on the ESQL routine's definition. Conversely, if the Java return type is *not void*, you must put a RETURNS clause on the ESQL routine's definition.
- Ensure that every parameter's type and direction matches the ESQL declaration, according to the rules listed in the table in "ESQL to Java data type mapping" on page 833.
- Ensure that the method's return type matches the data type of the RETURNS clause.
- Enclose EXTERNAL NAME in quotation marks because it must contain at least "class.method".
- If you want to invoke an overloaded Java method, you must create a separate ESQL definition for each overloaded method and give each ESQL definition a unique routine name.

You can use the Java User defined Node (UDN) API in your Java method, provided that you observe the restrictions documented in "Restrictions on Java routines" on page 834. For more information about using the UDN API, see [Compiling a Java user-defined node](#) .

Java routine example 1

This routine contains three parameters of varying directions, and returns an integer, which maps to a Java return type of `java.lang.Long`.

```
CREATE FUNCTION myProc1( IN P1 INTEGER, OUT P2 INTEGER, INOUT P3 INTEGER )
  RETURNS INTEGER
  LANGUAGE JAVA
  EXTERNAL NAME "com.ibm.broker.test.MyClass.myMethod1";
```

You can use the following ESQL to invoke `myProc1`:

```
CALL myProc1( intVar1, intVar2, intVar3) INTO intReturnVar3;
-- or
SET intReturnVar3 = myProc1( intVar1, intVar2, intVar3);
```

Java routine example 2

This routine contains three parameters of varying directions and has a Java return type of `void`.

```
CREATE PROCEDURE myProc2( IN P1 INTEGER, OUT P2 INTEGER, INOUT P3 INTEGER )
  LANGUAGE JAVA
  EXTERNAL NAME "com.ibm.broker.test.MyClass.myMethod2";
```

You must use the following ESQL to invoke `myProc2`:

```
CALL myProc2(intVar1, intVar2, intVar3);
```

The following Java class provides a method for each of the preceding Java examples:

```
package com.ibm.broker.test;

class MyClass {
  public static Long myMethod1( Long P1, Long[] P2 Long[] P3) { ... }
  public static void myMethod2( Long P2, Long[] P2 Long[] P3) { ... }
```

```

/* When either of these methods is called:
   P1 may or may not be NULL (depending on the value of intVar1).
   P2[0] is always NULL (whatever the value of intVar2).
   P3[0] may or may not be NULL (depending on the value of intVar3).
   This is the same as with LANGUAGE ESQL routines.
   When these methods return:
       intVar1 is unchanged
       intVar2 may still be NULL or may have been changed
       intVar3 may contain the same value or may have been changed.
   This is the same as with LANGUAGE ESQL routines.

   When myMethod1 returns: intReturnVar3 is either NULL (if the
   method returns NULL) or it contains the value returned by the
   method.
*/
}

```

ESQL to Java data type mapping

The following table summarizes the mappings from ESQL to Java.

Notes:

- Only the Java scalar wrappers are passed to Java.
- The ESQL scalar types are mapped to Java data types as object wrappers, or object wrapper arrays, depending upon the direction of the procedure parameter. Each wrapper array contains exactly one element.
- Scalar object wrappers are used to allow NULL values to be passed to and from Java methods.

ESQL data types ¹	Java IN data types	Java INOUT and OUT data types
INTEGER, INT	java.lang.Long	java.lang.Long []
FLOAT	java.lang.Double	java.lang.Double[]
DECIMAL	java.math.BigDecimal	java.math.BigDecimal[]
CHARACTER, CHAR	java.lang.String	java.lang.String[]
BLOB	byte[]	byte[][]
BIT	java.util.BitSet	java.util.BitSet[]
DATE	com.ibm.broker.plugin.MbDate	com.ibm.broker.plugin.MbDate[]
TIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
GMTTIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
TIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
GMTTIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
INTERVAL	Not supported	Not supported
BOOLEAN	java.lang.Boolean	java.lang.Boolean[]
REFERENCE (to a message tree) ^{3 4} _{5 6}	com.ibm.broker.plugin.MbElement	com.ibm.broker.plugin.MbElement[] (Supported for INOUT. Not supported for OUT)
ROW	Not supported	Not supported
LIST	Not supported	Not supported

1. Variables that are declared to be CONSTANT (or references to variables declared to be CONSTANT) are not allowed to have the direction INOUT or OUT.

2. The time zone set in the Java variable is not important; you obtain the required time zone in the output ESQL.
3. The reference parameter cannot be NULL when passed into a Java method.
4. The reference cannot have the direction OUT when passed into a Java method.
5. If an *MbElement* is passed back from Java to ESQL as an INOUT parameter, it must point to a location in the same message tree as that pointed to by the *MbElement* that was passed into the called Java method.

For example, if an ESQL reference to `OutputRoot.XML.Test` is passed into a Java method as an INOUT *MbElement*, but a different *MbElement* is passed back to ESQL when the call returns, the different element must also point to somewhere in the `OutputRoot` tree.

6. An *MbElement* cannot be returned from a Java method with the RETURNS clause, because no ESQL routine can return a reference. However, an *MbElement* can be returned as an INOUT direction parameter, subject to the conditions described in point 5 above.

A REFERENCE to a scalar variable can be used in the CALL of a Java method, provided that the data type of the variable the reference refers to matches the corresponding data type in the Java program signature.

Restrictions on Java routines

The following restrictions apply to Java routines called from ESQL:

- You must ensure that the Java method is threadsafe (reentrant).
- The only database connections permitted are JDBC type 4 connections. Furthermore, database operations are not part of a broker transaction; this means that they cannot be controlled by an external resource coordinator (such as would be the case in an XA environment).
- The Java User defined Node (UDN) API should be used only by the same thread that invoked the Java method.

You are allowed to spawn threads inside your method. However, spawned threads must not use the Java plug-in APIs and you must return control back to the broker.

Note that all restrictions that apply to the usage of the UDN API also apply to Java methods called from ESQL.

- Java methods called from ESQL must not use the `MbNode` class. This means that they cannot create objects of type `MbNode`, or call any of the methods on an existing `MbNode` object.
- If you want to perform MQ or JMS work inside a Java method called from ESQL, you must follow the User Defined Node (UDN) guidelines for performing MQ and JMS work in a UDN.

Deploying Java classes

We recommend that you deploy your Java classes inside a Java Archive (JAR) file. There are two ways to deploy a JAR file to the broker:

1. **By adding it to the Broker Archive (BAR) file**

This is the recommended method.

You can add a JAR file to the BAR file manually, by hand, or automatically, using the tooling. It is recommended that you use the tooling.

If the tooling finds the correct Java class inside a referenced Java project open in the workspace, it automatically compiles the Java class into a JAR file and

adds it to the BAR file. This is the same procedure that you follow to deploy a Java Compute node inside a JAR, as described in User-defined node classloading.

When deploying a JAR file from the tooling, a redeploy of the BAR file containing the JAR file causes the referenced Java classes to be reloaded by the flow that has been redeployed; as does stopping and restarting a message flow that references a Java class. Ensure that you stop and restart (or redeploy) all flows that reference the JAR file that you want to update. This avoids the problem of some flows running with the old version of the JAR file and other flows running with the new version.

Note that the tooling will only deploy a JAR file; it will not deploy a standalone Java class file.

2. By placing it in either of the following:

- a. The <Workpath>/shared-classes/ folder on the machine running the broker
 - b. The CLASSPATH environment variable on the machine running the broker
- This procedure must be done manually; you cannot use the tooling.

In this method, redeploying the message flow does not reload the referenced Java classes; neither does stopping and restarting the message flow. The only way to reload the classes in this case is to stop and restart the broker itself.

Note that although you can deploy a standalone Java class by placing it in one of the two locations above, it is still recommended that you use a JAR file instead.

To enable the broker to find a Java class, ensure that it is in one of the above locations. If the broker cannot find the specified class, it throws an exception.

Although you have the choices shown above when deploying the JAR file, it is recommended that you choose the first method (allowing the tooling to deploy the BAR file) because this provides the greatest flexibility when redeploying the JAR file.

Database routines

Database Routines are routines implemented as database stored procedures. Database routines have a LANGUAGE clause of DATABASE, and must have a routine type of PROCEDURE.

When writing stored procedures in languages like C, you must use NULL indicators to ensure that your procedure can process the data correctly.

Although the database definitions of a stored procedure will vary between the databases, the ESQL used to invoke them does not. The names given to parameters in the ESQL do not have to match the names they are given on the database side. However, the external name of the routine, including any package or container specifications, must match its defined name in the database.

The DYNAMIC RESULT SET clause is allowed only for database routines. It is required only if a stored procedure returns one or more result sets. The integer parameter to this clause must be 0 (zero) or more and specifies the number of result sets to be returned.

The optional RETURNS clause is required if a stored procedure returns a single scalar value.

The EXTERNAL NAME clause specifies the name by which the database knows the routine. This can be either a qualified or an unqualified name, where the qualifier is the name of the database schema in which the procedure is defined. If you do not provide a schema name, the database connection user name is used as the schema in which to locate the procedure. If the required procedure does not exist in this schema, you must provide an explicit schema name, either on the routine definition or on the CALL to the routine at runtime. For more information about dynamically choosing the schema which contains the routine, see the "CALL statement" on page 813. When a qualified name is used, the name must be in quotation marks.

A fully qualified routine normally takes the form:

```
EXTERNAL NAME "mySchema.myProc";
```

However, if the procedure belongs to an Oracle package, the package is treated as part of the procedure's name. Therefore you must provide a schema name as well as the package name, in the form:

```
EXTERNAL NAME "mySchema.myPackage.myProc";
```

This allows the schema, but not the package name, to be chosen dynamically in the CALL statement.

If the name of the procedure contains SQL wildcards (which are the percent (%) character and the underscore (_) character), the procedure name is modified by the broker to include the database escape character immediately before each wildcard character. This ensures that the database receives the wildcards as literal characters. For example, assuming that the database escape character is a backslash, the clause below is modified by the broker so that "mySchema.Proc__" is passed to the database. ;

```
EXTERNAL NAME "mySchema.Proc_";
```

All external procedures have the following restrictions:

- A stored procedure cannot be overloaded on the database side. A stored procedure is considered overloaded if there is more than one procedure of the same name in the same database schema. If the broker detects that a procedure has been overloaded, it raises an exception.
- Parameters cannot be of the ESQL REFERENCE, ROW, LIST, or INTERVAL data-types.
- User-defined types cannot be used as parameters or as return values.

Database routine example 1

The following is a simple ESQL definition of a stored procedure that returns a single scalar value and an OUT parameter:

```
CREATE PROCEDURE myProc1(IN P1 INT, OUT P2 INT)
LANGUAGE DATABASE
RETURNS INTEGER
EXTERNAL NAME "myschema.myproc";
```

Use this ESQL to invoke the myProc1 routine:

```
/*using CALL statement invocation syntax*/
CALL myProc1(intVar1, intVar2) INTO intReturnVar3;
```

```
/*or using function invocation syntax*/
SET intReturnVar3 = myProc1(intVar1, intVar2);
```

Database routine example 2

The following ESQL code demonstrates how to define and call DB2 stored procedures:

ESQL Definition:

```
DECLARE inputParm CHARACTER;
DECLARE outputParm CHARACTER;
DECLARE inputOutputParm CHARACTER;

SET inputParm = 'Hello';
SET inputOutputParm = 'World';
CALL swapParms( inputParm, outputParm, inputOutputParm );

CREATE PROCEDURE swapParms (
    IN parm1 CHARACTER,
    OUT parm2 CHARACTER,
    INOUT parm3 CHARACTER
) EXTERNAL NAME dbSwapParms;
```

To register this stored procedure with DB2, copy the following script to a file (for example, test1.sql)

```
-- DB2 Example Stored Procedure
DROP PROCEDURE dbSwapParms @
CREATE PROCEDURE dbSwapParms
( IN in_param CHAR(32),
  OUT out_param CHAR(32),
  INOUT inout_param CHAR(32))
LANGUAGE SQL
BEGIN
SET out_param = inout_param;
  SET inout_param = in_param;
END @
```

and execute:

```
db2 -td@ -vf test1.sql
```

from the DB2 command prompt.

Expect the following results from running this code:

- The value of the IN parameter does not (and cannot, by definition) change.
- The value of the OUT parameter becomes “World”.
- The value of the INOUT parameter changes to “Hello”.

Database routine example 3

The following ESQL code demonstrates how to define and call Oracle stored procedures:

ESQL Definition:

```
DECLARE inputParm CHARACTER;
DECLARE outputParm CHARACTER;
DECLARE inputOutputParm CHARACTER;

SET inputParm = 'Hello';
SET inputOutputParm = 'World';
CALL swapParms( inputParm, outputParm, inputOutputParm );

CREATE PROCEDURE swapParms (
    IN parm1 CHARACTER,
    OUT parm2 CHARACTER,
    INOUT parm3 CHARACTER
) EXTERNAL NAME dbSwapParms;
```


To register this stored procedure with Oracle, copy the following script to a file (for example, test1.sql)

```
CREATE OR REPLACE PROCEDURE dbSwapParams
( in_param IN VARCHAR2,
  out_param OUT VARCHAR2,
  inout_param IN OUT VARCHAR2 )
AS
BEGIN
    out_param := inout_param;
    inout_param := in_param;
END;
/
```

and execute:

```
sqlplus <userid>/<password> @test1.sql
```

Expect the following results from running this code:

- The value of the IN parameter does not (and cannot, by definition) change.
- The value of the OUT parameter becomes “World”.
- The value of the INOUT parameter changes to “Hello”.

Database routine example 4

The following ESQL code demonstrates how to define and call SQL Server stored procedures:

ESQL Definition:

```
DECLARE inputParm CHARACTER;
DECLARE outputParm CHARACTER;
DECLARE inputOutputParm CHARACTER;

SET inputParm = 'Hello';
SET inputOutputParm = 'World';
CALL swapParams( inputParm, outputParm, inputOutputParm );

CREATE PROCEDURE swapParams (
    IN parm1 CHARACTER,
    INOUT parm2 CHARACTER,
    INOUT parm3 CHARACTER
) EXTERNAL NAME dbSwapParams;
```

To register this stored procedure with SQLServer, copy the following script to a file (for example, test1.sql)

```
|
| -- SQLServer Example Stored Procedure
| DROP PROCEDURE dbSwapParams
| go
| CREATE PROCEDURE dbSwapParams
|     @in_param    CHAR(32),
|     @out_param   CHAR(32) OUT,
|     @inout_param CHAR(32) OUT
| AS
|     SET NOCOUNT ON
|     SET @out_param = @inout_param
|     SET @inout_param = @in_param
| go
```

and execute:

```
isql -U<userid> -P<password> -S<server> -d<datasource> -itest1.sql
```

Note:

1. SQL Server considers OUTPUT parameters from stored procedures as INPUT/OUTPUT parameters.
If you declare these as OUT parameters in your ESQL you encounter a type mismatch error at run time. To avoid that mismatch you must declare SQL Server OUTPUT parameters as INOUT in your ESQL.
2. You should use the SET NOCOUNT ON option, as shown in the preceding example, with SQL Stored Procedures for the following reasons:
 - a. To limit the amount of data returned from SQLServer to the broker.
 - b. To allow result-sets to be returned correctly.

Expect the following results from running this code:

- The value of the IN parameter does not (and cannot, by definition) change.
- The value of the OUT parameter becomes "World".
- The value of the INOUT parameter changes to "Hello".

Database routine example 5

The following ESQL code demonstrates how to define and call SYBASE stored procedures:

```

ESQL Definition:
DECLARE inputParm CHARACTER;
DECLARE outputParm CHARACTER;
DECLARE inputOutputParm CHARACTER;

SET inputParm = 'Hello';
SET inputOutputParm = 'World';
CALL swapParms( inputParm, outputParm, inputOutputParm );

CREATE PROCEDURE swapParms (
    IN parm1 CHARACTER,
    INOUT parm2 CHARACTER,
    INOUT parm3 CHARACTER
) EXTERNAL NAME dbSwapParms;

```

To register this stored procedure with SYBASE, copy the following script to a file (for example, test1.sql)

```

-- SYBASE Example Stored Procedure
DROP PROCEDURE dbSwapParms
go
CREATE PROCEDURE dbSwapParms
    @in_param CHAR(32),
    @out_param CHAR(32) OUT,
    @inout_param CHAR(32) OUT
AS
    SET @out_param = @inout_param
    SET @inout_param = @in_param
go

```

and execute:

```
isql -U<userid> -P<password> -S<server> -d<datasource> -itest1.sql
```

Note: SYBASE considers OUTPUT parameters from stored procedures as INPUT/OUTPUT parameters.

If you declare these as OUT parameters in your ESQL you encounter a type mismatch error at run time. To avoid that mismatch you must declare SYBASE OUTPUT parameters as INOUT in your ESQL.

|
|
|
|

Expect the following results from running this code:

- The value of the IN parameter does not (and cannot, by definition) change.
- The value of the OUT parameter becomes "World".
- The value of the INOUT parameter changes to "Hello".

Database routine example 6

This example shows how to call a stored procedure that returns two result sets, as well as an out parameter:

```
CREATE PROCEDURE myProc1 (IN P1 INT, OUT P2 INT)
  LANGUAGE DATABASE
  DYNAMIC RESULT SETS 2
  EXTERNAL NAME "myschema.myproc";
```

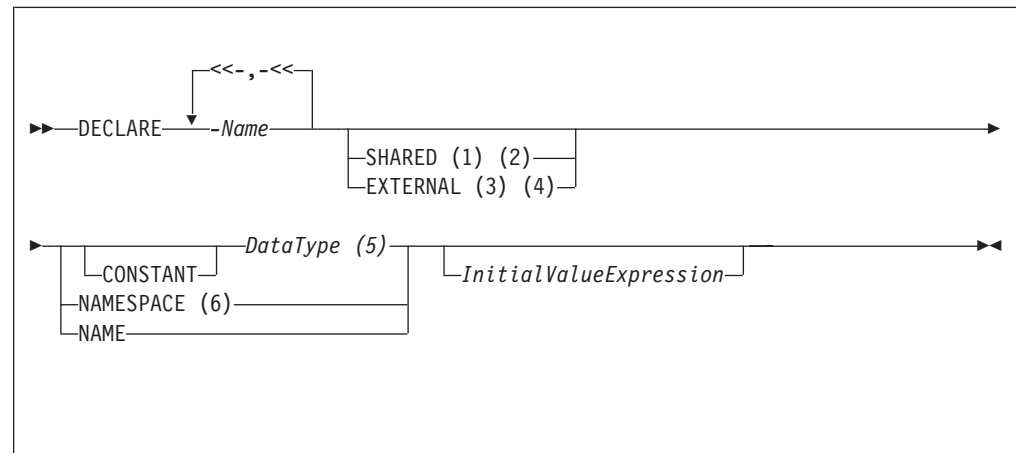
Use the following ESQL to invoke myProc1:

```
/* using a field reference */
CALL myProc1(intVar1, intVar2, Environment.RetVal[], OutputRoot.XML.A[])
/* using a reference variable*/
CALL myProc1(intVar1, intVar2, myReferenceVariable.RetVal[], myRef2.B[])
```

DECLARE statement

The DECLARE statement defines a variable, the data type of the variable and, optionally, its initial value.

Syntax



1. The SHARED keyword is not allowed within a function or procedure.
2. You cannot specify SHARED with a *DataType* of REFERENCE. (To store a message tree in a shared variable, use the ROW data type.)
3. EXTERNAL variables are implicitly constant.
4. You are recommended to give an EXTERNAL variable an initial value.
5. If you specify a *DataType* of REFERENCE, you must specify an initial value (of either a variable or a tree) in *InitialValueExpression*.
6. When the NAMESPACE and NAME clauses are used the values are implicitly constant and of type CHARACTER.

Types of variable

You can use the "DECLARE statement" to define three types of variable:

External

External variables (defined with the EXTERNAL keyword) are also known as *user-defined properties* (UDPs): see “User-defined properties in ESQL” on page 149. They exist for the entire lifetime of a message flow and are visible to all messages passing through the flow. Their initial values (optionally set by the DECLARE statement) can be modified, at design time, by the Message Flow editor, or, at deployment time, by the BAR editor. Their values cannot be modified by ESQL.

Normal

“Normal” variables have a lifetime of just one message passing through a node. They are visible to that message only. To define a “normal” variable, omit both the EXTERNAL and SHARED keywords.

Shared

Shared variables can be used to implement an in-memory cache in the message flow, see “Optimizing message flow response times” on page 73. Shared variables have a long lifetime and are visible to multiple messages passing through a flow, see “Long-lived variables” on page 150. They exist for the lifetime of the execution group process, the lifetime of the flow or node, or the lifetime of the node’s SQL that declares the variable (whichever is the shortest). They are initialized when the first message passes through the flow or node after each broker start up.

See also the ATOMIC option of the “BEGIN ... END statement” on page 807. The BEGIN ATOMIC construct is useful when a number of changes need to be made to a shared variable and it is important to prevent other instances seeing the intermediate states of the data.

CONSTANT

Use CONSTANT to define a constant. You can declare constants within schemas, modules, routines, or compound statements (both implicit and explicit). The behavior of these cases is as follows:

- Within a compound statement, constants and variables occupy the same namespace.
- Within expressions, a constant or variable declared within a compound statement overlays all constants and variables of the same name declared in containing compound statements, module and schema.
- Within field reference namespace fields, a namespace constant declared within a compound statement overlays all namespace constants of the same name declared in containing compound statements, and similarly for name constants.

A constant or variable declared within a routine overlays any parameters of the same name, and all constants and variables of the same name declared in a containing module or schema.

DataType

The possible values that you can specify for *DataType* are:

- BOOL
- BOOLEAN
- INT
- INTEGER
- FLOAT
- DEC
- DECIMAL

- DATE
- TIME
- TIMESTAMP
- GMTTIME
- GMTTIMESTAMP
- INTERVAL: does not apply to external variables (EXTERNAL option specified)
- CHAR
- CHARACTER
- BLOB
- BIT
- ROW: does not apply to external variables (EXTERNAL option specified)
- REF: does not apply to external or shared variables (EXTERNAL or SHARED option specified)
- REFERENCE-TO: does not apply to external or shared variables (EXTERNAL or SHARED option specified)

Note: If you specify a *DataType* of REFERENCE, you must also specify *InitialValueExpression*.

EXTERNAL

Use EXTERNAL to denote a user-defined property (UDP). A UDP is a user-defined constant whose initial value (optionally set by the DECLARE statement) can be modified, at design time, by the Message Flow editor, or overridden, at deployment time, by the Broker Archive editor. Its value cannot be modified by ESQL.

For an overview of UDPs, see “User-defined properties in ESQL” on page 149.

When a UDP is given an initial value on the DECLARE statement this becomes its default. However, any value specified by the Message Flow editor at design time, or by the BAR editor at deployment time (even a zero length string) overrides any initial value coded on the DECLARE statement.

All UDPs in a message flow must have a value, given either on the DECLARE statement or by the Message Flow or BAR editor; otherwise a deployment-time error occurs. At run time, after the UDP has been declared its value can be queried by subsequent ESQL statements but not modified.

The advantage of UDPs is that their values can be changed by operational staff at deployment time. If, for example, you use the UDPs to hold configuration data, it means that you can configure a message flow for a particular machine, task, or environment at deployment time, without having to change the code at the node level.

You can declare UDPs only in modules or schemas.

The following types of broker node are capable of accessing UDPs:

- Compute
- Database
- Filter
- Nodes derived from these node-types

Take care when specifying the data type of a UDP, because a CAST occurs to cast to the requested *DataType*.

Example 1

```
DECLARE mycolour EXTERNAL CHARACTER 'blue';
```

Example 2

```
DECLARE TODAYSCOLOR EXTERNAL CHARACTER;  
SET COLOR = TODAYSCOLOR;
```

where TODAYSCOLOR is a user-defined property that has a TYPE of CHARACTER and a VALUE set by the Message Flow Editor.

NAME

Use NAME to define an alias (another name) by which a variable can be known.

Example 1

```
-- The following statement gives Schema1 an alias of 'Joe'.  
DECLARE Schema1 NAME 'Joe';  
-- The following statement produces a field called 'Joe'.  
SET OutputRoot.XML.Data.Schema1 = 42;  
  
-- The following statement inserts a value into a table called Table1  
-- in the schema called 'Joe'.  
INSERT INTO Database.Schema1.Table1 (Answer) VALUES 42;
```

Example 2

```
DECLARE Schema1 EXTERNAL NAME;  
  
CREATE FIRSTCHILD OF OutputRoot.XML.TestCase.Schema1 Domain('XML')  
    NAME 'Node1' VALUE '1';  
  
-- If Schema1 has been given the value 'red', the result would be:  
<xml version="1.0"?>  
<TestCase>  
  <red>  
    <Node1>1</Node1>  
  </red>
```

NAMESPACE

Use NAMESPACE to define an alias (another name) by which a namespace can be known.

Example

This example illustrates a namespace declaration, its use as a *SpaceId* in a path, and its use as a character constant in a namespace expression:

```
DECLARE prefixOne NAMESPACE 'http://www.example.com/P01';  
  
-- On the right hand side of the assignment a namespace constant  
-- is being used as such while, on the left hand side, one is  
-- being used as an ordinary constant (that is, in an expression).  
  
SET OutputRoot.XML.{prefixOne}:{'PurchaseOrder'} =  
    InputRoot.XML.prefixOne:PurchaseOrder;
```

SHARED

Use SHARED to define a shared variable. Shared variables are private to the flow (if declared within a schema) or node (if declared within a module) but are shared

between instances of the flow (threads). There is no type of variable that is visible more widely than at the flow level. For example, you cannot share variables across execution groups.

Shared variables can be used to implement an in-memory cache in the message flow, see “Optimizing message flow response times” on page 73. Shared variables have a long lifetime and are visible to multiple messages passing through a flow, see “Long-lived variables” on page 150. They exist for the lifetime of the execution group process, the lifetime of the flow or node, or the lifetime of the node’s SQL that declares the variable (whichever is the shortest). They are initialized when the first message passes through the flow or node after each broker start up.

You cannot define a shared variable within a function or procedure.

The advantages of shared variables, relative to a databases, are that:

- Write access is very much faster.
- Read access to small data structures is faster.
- Access is direct. That is, there is no need to use a special function (SELECT) to get data, or special statements (INSERT, UPDATE, or DELETE) to modify data. Instead, you can refer to the data directly in expressions.

The advantages of databases, relative to shared variables, are that:

- The data is persistent.
- The data is changed transactionally.

These read-write variables, with a life greater than that of one message but which perform better than a database, are ideal for users prepared to sacrifice the persistence and transactional advantages of databases in order to obtain better performance.

With flow-shared variables (that is, those defined at the schema level), take care when multiple flows can update the variables, especially if the variable is being used as a counter. Likewise, with node-shared variables (that is, those defined at the module level), take care when multiple instances can update the variables.

Shared row variables allow a user program to make an efficient read/write copy of an input node’s message. This is generally useful and, in particular, simplifies the technique for handling large messages.

“There is a restriction that subtrees cannot be directly copied from one shared row variable to another shared row variable. Subtrees can be *indirectly* copied by using a non-shared row variable. Scalar values extracted from one shared row variable (using the FIELDVALUE function) can be copied to another shared row variable.

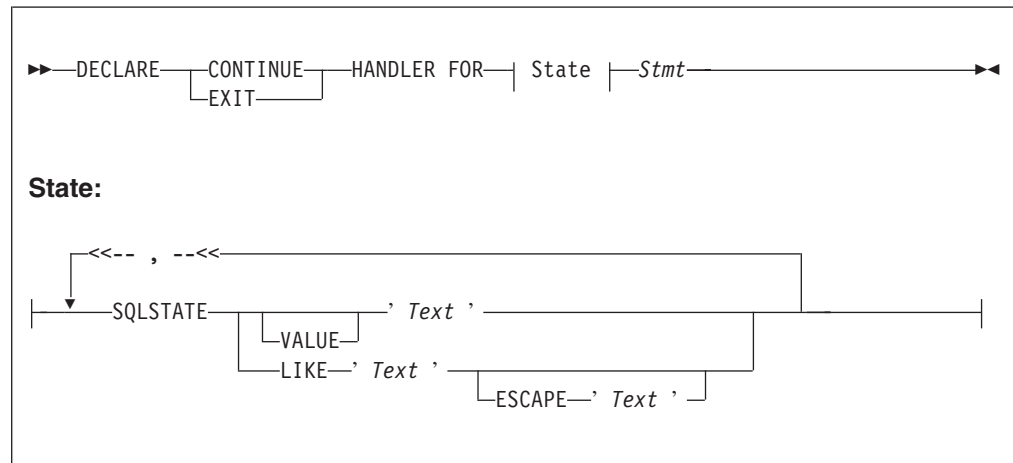
Example

For an example of the use of shared variables, see the “Message routing” sample program, which shows how to use both shared and external variables. The “Message routing” sample is in the Samples Gallery in the Message Brokers Toolkit.

DECLARE HANDLER statement

The DECLARE HANDLER statement creates an error handler for handling exceptions.

Syntax



You can declare handlers in both explicitly declared (BEGIN...END) scopes and implicitly declared scopes (for example, the ELSE clause of an IF statement). However, all handler declarations must be together at the top of the scope, before any other statements.

If there are no exceptions, the presence of handlers has no effect on the behavior or performance of an SQL program. If an exception occurs, WebSphere Message Broker compares the SQL state of the exception with the SQL states associated with any relevant handlers, until either the exception leaves the node (just as it would if there were no handlers) or a matching handler is found. Within any one scope, handlers are searched in the order they are declared; that is, first to last. Scopes are searched from the innermost to outermost.

The SQL state values provided in DECLARE... HANDLER... statements can be compared directly with the SQL state of the exception or can be compared using wild card characters. To compare the state values directly, specify either VALUE or no condition operator. To make a wild card comparison, use the underscore and percent characters to represent single and multiple character wild cards, respectively, and specify the LIKE operator. The wild card method allows all exceptions of a general type to be handled without having to list them exhaustively.

If a matching handler is found, the SQLSTATE and other special registers are updated (according to the rules described below) and the handler's statement is processed.

As the handler's statement must be a single statement, it is typically a compound statement (such as BEGIN...END) that contains multiple other statements. There is no special behavior associated with these inner statements and there are no special restrictions. They can, for example, include RETURN, ITERATE, or LEAVE; these affect their containing routines and looping constructs in the same way as if they were contained in the scope itself.

Handlers can contain handlers for exceptions occurring within the handler itself

If processing of the handler's code completes without throwing further unhandled exceptions, execution of the normal code is resumed as follows:

- For EXIT handlers, the next statement processed is the first statement after the handler's scope.
- For CONTINUE handlers, it is the first directly-contained statement after the one that produced the exception.

Each handler has its own `SQLCODE`, `SQLSTATE`, `SQLNATIVEERROR`, and `SQLERRORTXT` special registers. These come into scope and their values are set just before the handler's first statement is executed. They remain valid until the handler's last statement has been executed. Because there is no carry over of `SQLSTATE` values from one handler to another, handlers can be written independently.

Handlers absorb exceptions, preventing their reaching the input node and thus causing the transaction to be committed rather than rolled back. A handler can use a `RESIGNAL` or `THROW` statement to prevent this.

Example

```
-- Drop the tables so that they can be recreated with the latest definition.
-- If the program has never been run before, errors will occur because you
-- can't drop tables that don't exist. We ignore these.
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE LIKE '%' BEGIN END;

  PASSTHRU 'DROP TABLE Shop.Customers' TO Database.DSN1;
  PASSTHRU 'DROP TABLE Shop.Invoices' TO Database.DSN1;
  PASSTHRU 'DROP TABLE Shop.Sales' TO Database.DSN1;
  PASSTHRU 'DROP TABLE Shop.Parts' TO Database.DSN1;
END;
```

Related concepts

"ESQL overview" on page 146

Related tasks

"Developing ESQL" on page 145

Related reference

"RESIGNAL statement" on page 878

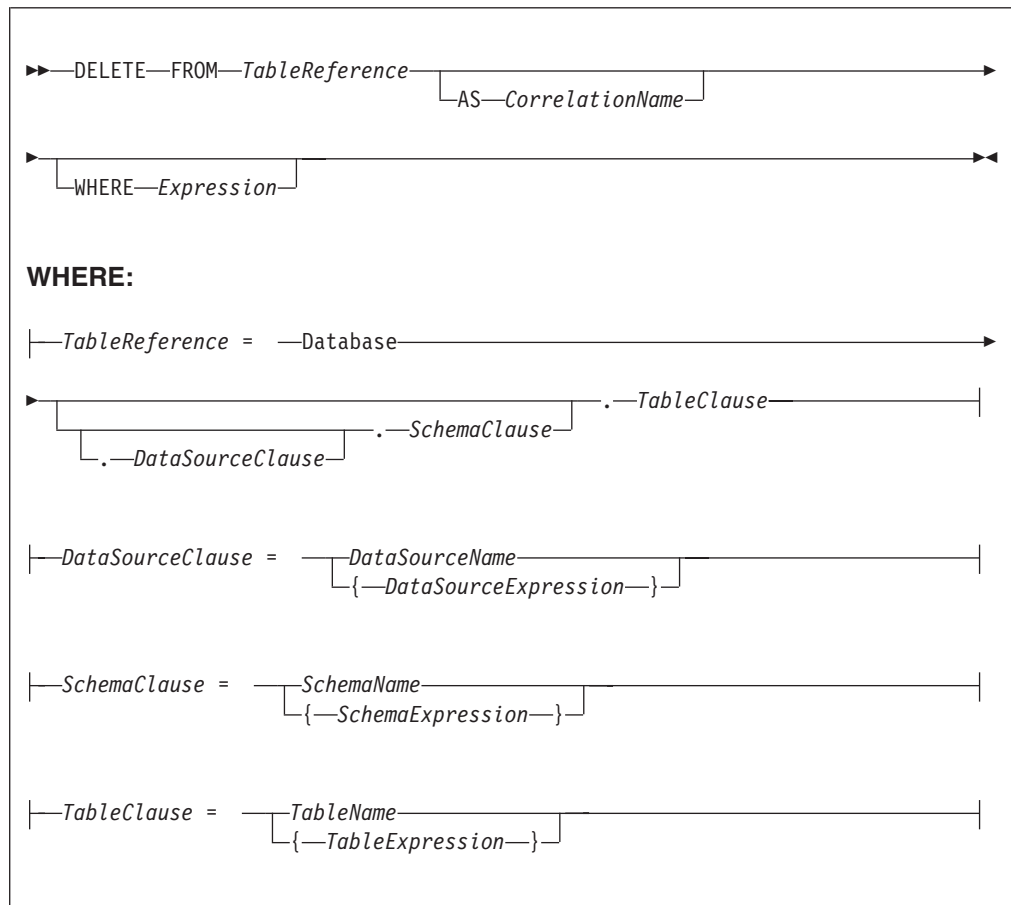
"Syntax diagrams: available types" on page 780

"ESQL statements" on page 804

DELETE FROM statement

The `DELETE FROM` statement deletes rows from a table in an external database, based on a search condition.

Syntax



All rows for which the WHERE clause expression evaluates to TRUE are deleted from the table identified by *TableReference*.

Each row is examined in turn and a variable is set to point to the current row. Typically, the WHERE clause expression uses this variable to access column values and thus cause rows to be retained or deleted according to their contents. The variable is referred to by *CorrelationName* or, in the absence of an AS clause, by *TableName*.

Table reference

A table reference is a special case of the field references used to refer to message trees. It always starts with the word "Database" and may contain any of the following:

- A table name only
- A schema name and a table name
- A data source name (that is, the name of a database instance), a schema name, and a table name

In each case, the name may be specified directly or by an expression enclosed in braces ({...}). A directly-specified data source, schema, or table name is subject to name substitution. That is, if the name used has been declared to be a known name, the value of the declared name is used rather than the name itself (see "DECLARE statement" on page 851).

If a schema name is not specified, the default schema for the broker's database user is used.

If a data source name is not specified, the database pointed to by the node's data source attribute is used.

The WHERE clause

The WHERE clause expression can use any of the broker's operators and functions in any combination. It can refer to table columns, message fields, and any declared variables or constants.

However, be aware that the broker treats the WHERE clause expression by examining the expression and deciding whether the whole expression can be evaluated by the database. If it can, it is given to the database. In order to be evaluated by the database, it must use only those functions and operators supported by the database.

The WHERE clause can, however, refer to message fields, correlation names declared by containing SELECTs, and to any other declared variables or constants within scope.

If the whole expression cannot be evaluated by the database, the broker looks for top-level AND operators and examines each sub-expression separately. It then attempts to give the database those sub-expressions that it can evaluate, leaving the broker to evaluate the rest. You need to be aware of this situation for two reasons:

1. Apparently trivial changes to WHERE clause expressions can have large effects on performance. You can determine how much of the expression was given to the database by examining a user trace.
2. Some databases' functions exhibit subtle differences of behavior from those of the broker.

Handling errors

It is possible for errors to occur during delete operations. For example, the database may not be operational. In these cases, an exception is thrown (unless the node has its `throw exception on database error` property set to `FALSE`). These exceptions set appropriate SQL code, state, native error, and error text values and can be dealt with by error handlers (see the `DECLARE HANDLER` statement).

For further information about handling database errors, see "Capturing database state" on page 219.

Examples

The following example assumes that the `dataSource` property has been configured and that the database it identifies has a table called `SHAREHOLDINGS`, with a column called `ACCOUNTNO`.

```
DELETE FROM Database.SHAREHOLDINGS AS S
      WHERE S.ACCOUNTNO = InputBody.AccountNumber;
```

This removes all the rows from the `SHAREHOLDINGS` table where the value in the `ACCOUNTNO` column (in the table) is equal to that in the `AccountNumber` field in the message. This may delete zero, one, or more rows from the table.

The next example shows the use of calculated data source, schema, and table names:

```
-- Declare variables to hold the data source, schema, and table names and
-- set their default values
DECLARE Source CHARACTER 'Production';
DECLARE Schema CHARACTER 'db2admin';
DECLARE Table CHARACTER 'DynamicTable1';

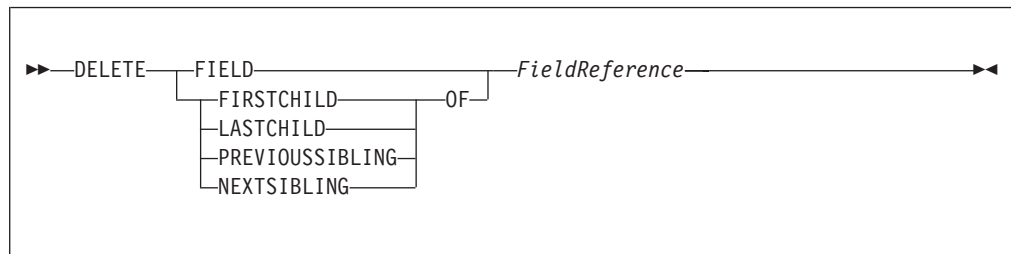
-- Code which calculates their actual values comes here

-- Delete rows from the table
DELETE FROM Database.{Source}.{Schema}.{Table} As R WHERE R.Name = 'Joe';
```

DELETE statement

The DELETE statement detaches and destroys a portion of a message tree, allowing its memory to be reused. This statement is particularly useful when handling very large messages.

Syntax



If the target field does not exist, the statement does nothing and normal processing continues. If any reference variables point into the deleted portion, they are disconnected from the tree so that no action involving them has any effect, and the LASTMOVE function returns FALSE. Disconnected reference variables can be reconnected by using a MOVE... TO... statement.

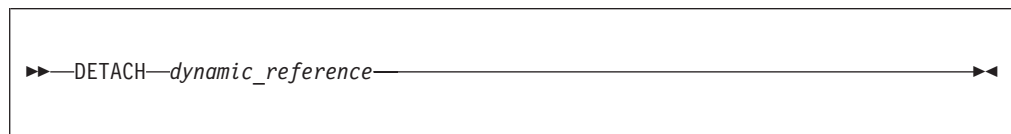
Example

```
DELETE FIELD OutputRoot.XML.Data.Folder1.Folder12;
DELETE LASTCHILD OF Cursor;
```

DETACH statement

The DETACH statement detaches a portion of a message tree without deleting it. This portion can be reattached using the ATTACH statement.

Syntax



For information about dynamic references, see “Creating dynamic field references” on page 179.

For an example of DETACH, see the example in “ATTACH statement” on page 806.

EVAL statement

The EVAL statement takes a character value, interprets it as an SQL statement, and processes it.

The EVAL function (also described here) takes a character value, but interprets it as an ESQL expression that returns a value.

Note: User defined functions and procedures cannot be defined within an EVAL statement or EVAL function.

Syntax

```
►►—EVAL—( SQL_character_value )—————►►
```

EVAL takes one parameter in the form of an expression, evaluates this expression, and casts the resulting value to a character string if it is not one already. The expression that is passed to EVAL must therefore be able to be represented as a character string.

After this first stage evaluation is complete, the behavior of EVAL depends on whether it is being used as a complete ESQL statement, or in place of an expression that forms part of an ESQL statement:

- If it is a complete ESQL statement, the character string derived from the first stage evaluation is executed as if it were an ESQL statement.
- If it is an expression that forms part of an ESQL statement, the character string is evaluated as if it were an ESQL expression and EVAL returns the result.

In the following examples, A and B are integer scalar variables, and scalarVar1 and OperatorAsString are character string scalar variables.

The following examples are valid uses of EVAL:

- SET OutputRoot.XML.Data.Result = EVAL(A+B);

The expression A+B is acceptable because, although it returns an integer value, integer values are representable as character strings, and the necessary cast is performed before EVAL continues with its second stage of evaluation.

- SET OutputRoot.XML.Data.Result = EVAL('A' || operatorAsString || 'B');
- EVAL('SET ' || scalarVar1 || ' = 2;');

The semicolon included at the end of the final string literal is necessary, because if EVAL is being used in place of an ESQL statement, its first stage evaluation must return a string that represents a valid ESQL statement, including the terminating semicolon.

Variables declared within an EVAL statement do not exist outside that EVAL statement. In this way EVAL is similar to a function, in which locally-declared variables are local only, and go out of scope when the function is exited.

The real power of EVAL is that it allows you to dynamically construct ESQL statements or expressions. In the second and third examples above, the value of scalarVar1 or operatorAsString can be set according to the value of an incoming message field, or other dynamic value, allowing you to effectively control what ESQL is executed without requiring a potentially lengthy IF THEN ladder.

However, consider the performance implications in using EVAL. Dynamic construction and execution of statements or expressions is necessarily more time-consuming than simply executing pre-constructed ones. If performance is vital, you might prefer to write more specific, but faster, ESQL.

The following are not valid uses of EVAL:

- SET EVAL(scalarVar1) = 2;
In this example, EVAL is being used to replace a field reference, not an expression.
- SET OutputRoot.XML.Data.Result[] = EVAL((SELECT T.x FROM Database.y AS T));
In this example, the (SELECT T.x FROM Database.y) passed to EVAL returns a list, which is not representable as a character string.

The following example is acceptable because (SELECT T.x FROM Database.y AS T) is a character string literal, not an expression in itself, and therefore is representable as a character string.

```
SET OutputRoot.XML.Data.Result[]
= EVAL('(SELECT T.x FROM Database.y AS T)');
```

FOR statement

The FOR statement iterates through a list (for example, a message array).

Syntax

```
►►—FOR—correlation_name—AS—field_reference—DO—statements—END—FOR—◄◄
```

For each iteration, the FOR statement makes the correlation variable (*correlation_name* in the syntax diagram) equal to the current member of the list (*field_reference*) and then executes the block of statements. The advantage of the FOR statement is that it iterates through a list without your having to write any sort of loop construct (and eliminates the possibility of infinite loops).

For example the following ESQL:

```
SET OutputRoot.MQMD=InputRoot.MQMD;

SET Environment.SourceData.Folder[1].Field1 = 'Field11Value';
SET Environment.SourceData.Folder[1].Field2 = 'Field12Value';
SET Environment.SourceData.Folder[2].Field1 = 'Field21Value';
SET Environment.SourceData.Folder[2].Field2 = 'Field22Value';

DECLARE i INTEGER 1;
FOR source AS Environment.SourceData.Folder[] DO
    CREATE LASTCHILD OF OutputRoot.XML.Data.ResultData.MessageArrayTest.Folder[i]
        NAME 'FieldA' VALUE '\ ' || source.Field1 || '\ ' || CAST(i AS CHAR);
```

```

        CREATE LASTCHILD OF OutputRoot.XML.Data.ResultData.MessageArrayTest.Folder[i]
            NAME 'FieldB' VALUE '\' || source.Field2 || '\' || CAST(i AS CHAR);
    SET i = i + 1;
END FOR;

```

generates the output message:

```

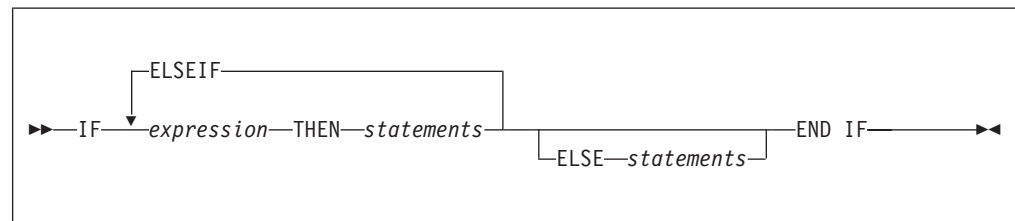
<Data>
  <ResultData>
    <MessageArrayTest>
      <Folder>
        <FieldA>Field11Value1</FieldA>
        <FieldB>Field12Value1</FieldB>
      </Folder>
      <Folder>
        <FieldA>Field21Value2</FieldA>
        <FieldB>Field22Value2</FieldB>
      </Folder>
    </MessageArrayTest>
  </ResultData>
</Data>

```

IF statement

The IF statement executes one set of statements based on the result of evaluating condition expressions.

Syntax



Each expression is evaluated in turn until one results in TRUE; the corresponding set of statements is then executed. If none of the expressions returns TRUE, and the optional ELSE clause is present, the ELSE clause's statements are executed.

UNKNOWN and FALSE are treated the same: the next condition expression is evaluated. ELSEIF is one word with no space between the ELSE and the IF. However, you can nest an IF statement within an ELSE clause: if you do, you can terminate both statements with END IF.

Example

```

IF i = 0 THEN
    SET size = 'small';
ELSEIF i = 1 THEN
    SET size = 'medium';
ELSEIF j = 4 THEN
    SET size = 'large';
ELSE
    SET size = 'unknown';
END IF;

```

```

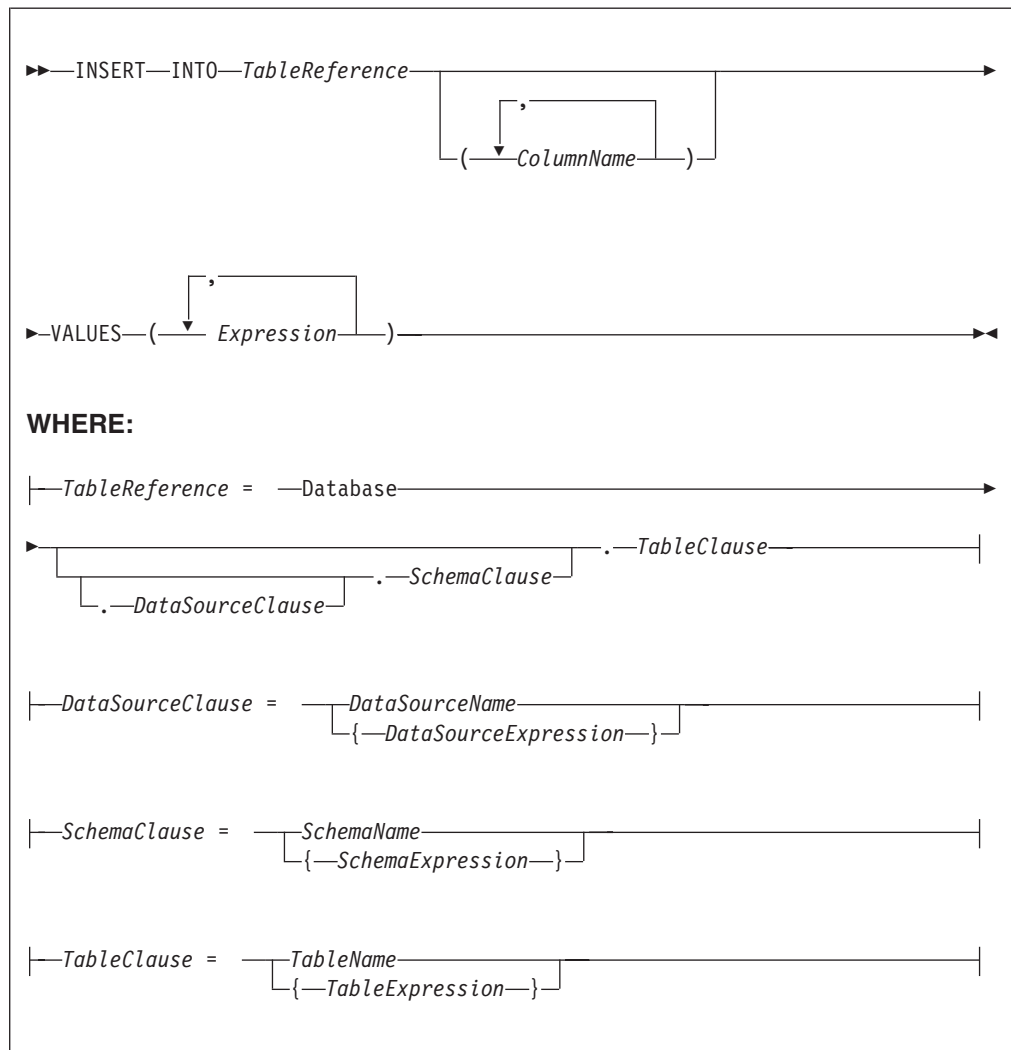
IF J > MAX THEN
  SET J = MAX;
  SET Limit = TRUE;
END IF;

```

INSERT statement

The INSERT statement inserts a row into a database table.

Syntax



A single row is inserted into the table identified by *TableReference*. The *ColumnName* list identifies those columns in the target table that are to be given specific values. These values are determined by the expressions within the VALUES clause (the first expression gives the value of the first named column, and so on). The number of expressions in the VALUES clause must be the same as the number of named columns. Any columns present in the table but not mentioned in the list are given their default values.

Table reference

A table reference is a special case of the field references used to refer to message trees. It always starts with the word “Database” and may contain any of the following:

- A table name only
- A schema name and a table name
- A data source name (that is, the name of a database instance), a schema name, and a table name

In each case, the name may be specified directly or by an expression enclosed in braces ({...}). A directly-specified data source, schema, or table name is subject to name substitution. That is, if the name used has been declared to be a known name, the value of the declared name is used rather than the name itself (see “DECLARE statement” on page 851).

If a schema name is not specified, the default schema for the broker’s database user is used.

If a data source name is not specified, the database pointed to by the node’s data source attribute is used.

Handling errors

It is possible for errors to occur during insert operations. For example, the database may not be operational, or the table may have constraints defined which the new row would violate. In these cases, an exception is thrown (unless the node has its *throw exception on database error* property set to FALSE). These exceptions set appropriate SQL code, state, native error, and error text values and can be dealt with by error handlers (see the DECLARE HANDLER statement).

For further information about handling database errors, see “Capturing database state” on page 219.

Examples

The following example assumes that the dataSource property of the Database node has been configured, and that the database it identifies has a table called TABLE1 with columns A, B, and C.

Given a message with the following generic XML body:

```
<A>
  <B>1</B>
  <C>2</C>
  <D>3</D>
</A>
```

The following INSERT statement inserts a new row into the table with the values 1, 2, and 3 for the columns A, B, and C:

```
INSERT INTO Database.TABLE1(A, B, C) VALUES (Body.A.B, Body.A.C, Body.A.D);
```

The next example shows the use of calculated data source, schema, and table names:

```
-- Declare variables to hold the data source, schema, and table names
-- and set their default values
DECLARE Source CHARACTER 'Production';
DECLARE Schema CHARACTER 'db2admin';
```

```

DECLARE Table CHARACTER 'DynamicTable1';

-- Code which calculates their actual values comes here

-- Insert the data into the tabl
INSERT INTO Database.{Source}.{Schema}.{Table} (Name, Value) values ('Joe', 12.34);

```

ITERATE statement

The ITERATE statement stops the current iteration of the containing WHILE, REPEAT, LOOP, or BEGIN statement identified by Label.

The containing statement evaluates its loop condition (if any), and either starts the next iteration or stops looping, as the condition dictates.

Syntax

```

▶▶—ITERATE—Label————▶▶

```

Example

In the following example, the loop iterates four times; that is the line identified by the comment Some statements 1 is passed through four times. However, the line identified by the comment Some statements 2 is passed through twice only because of the action of the IF and ITERATE statements. The ITERATE statement does **not** bypass testing the loop condition. Take particular care that the action of the ITERATE does not bypass the logic that makes the loop advance and eventually terminate. The loop count is incremented at the start of the loop in this example:

```

DECLARE i INTEGER;
SET i = 0;
X : REPEAT
  SET i = i + 1;

  -- Some statements 1

  IF i IN(2, 3) THEN
    ITERATE X;
  END IF;

  -- Some statements 2

UNTIL
  i >= 4
END REPEAT X;

```

ITERATE statements do not have to be directly contained by their labelled statement, making ITERATE statements particularly powerful.

LEAVE statement

The LEAVE statement stops the current iteration of the containing WHILE, REPEAT, LOOP, or BEGIN statement identified by Label.

The containing statement's evaluation of its loop condition (if any) is bypassed and looping stops.

Syntax

```
▶▶—LEAVE—Label—◀◀
```

Examples

In the following example, the loop iterates four times:

```
DECLARE i INTEGER;
SET i = 1;
X : REPEAT
  ...
  IF i >= 4 THEN
    LEAVE X;
  END IF;

  SET i = i + 1;
UNTIL
  FALSE
END REPEAT;
```

LEAVE statements do not have to be directly contained by their labelled statement, making LEAVE statements particularly powerful.

```
DECLARE i INTEGER;
SET i = 0;
X : REPEAT                                -- Outer loop
  ...
  DECLARE j INTEGER;
  SET j = 0;
  REPEAT                                  -- Inner loop
    ...
    IF i >= 2 AND j = 1 THEN
      LEAVE X;                            -- Outer loop left from within inner loop
    END IF;
    ...
    SET j = j + 1;
  UNTIL
    j >= 3
  END REPEAT;

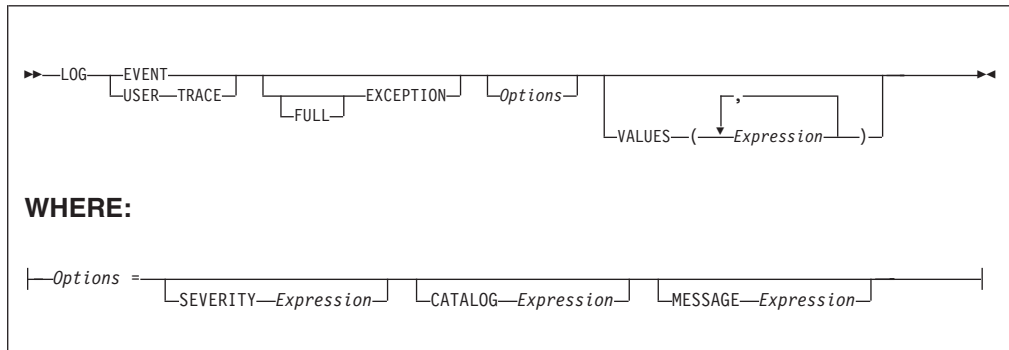
  SET i = i + 1;
UNTIL
  i >= 3
END REPEAT X;

-- Execution resumes here after the leave
```

LOG statement

The LOG statement writes a record to the event or user trace logs.

Syntax



CATALOG

CATALOG is an optional clause; if you omit it, it defaults to the WebSphere Message Broker current version catalog. To use the current WebSphere Message Broker version message catalog explicitly, use BIPV600 on all operating systems.

EVENT

A record is written to the event log (and also to user trace if user tracing is enabled).

EXCEPTION

The current exception (if any) is logged.

FULL

The complete nested exception report is logged (just as if the exception had reached the input node). If FULL is not specified, any wrapping exceptions are ignored and only the original exception is logged. Thus you can have a full report or simply the actual error report without the extra information concerning what was going on at the time. Note that a current exception only exists within handler blocks (see "Handling errors in message flows" on page 111).

MESSAGE

The number of the message to be used. If specified, the MESSAGE clause can contain any expression that returns a non-NULL, integer, value.

If you omit MESSAGE, its value defaults to the first message number (2951) in a block of messages provided for use by the LOG and THROW statements in the WebSphere Business Integration Message Broker catalog. If you enter a message number, you can use message numbers 2951 to 2999. Alternatively, you can generate your own catalog.

SEVERITY

The severity associated with the message. If specified, the SEVERITY clause can contain any expression that returns a non-NULL, integer, value. If you omit the clause, its value defaults to 1.

USER TRACE

A record is written to the user trace, whether user trace is enabled or not.

VALUES

Use the optional VALUES clause to provide values for the data inserts in your message. You can insert any number of pieces of information, but the messages supplied (2951 - 2999) cater for ten inserts only.

Note the general similarity of the LOG statement to the THROW statement.

```

-- Write a message to the event log specifying the severity, catalogue and message
-- number. Four inserts are provided
LOG EVENT SEVERITY 1 CATALOG 'BIPv600' MESSAGE 2951 VALUES(1,2,3,4);

-- Write to the trace log whenever a divide by zero occurs
BEGIN
  DECLARE a INT 42;
  DECLARE b INT 0;
  DECLARE r INT;

  BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE LIKE 'S22012' BEGIN
      LOG USER TRACE EXCEPTION VALUES(SQLSTATE, 'DivideByZero');

      SET r = 0x7FFFFFFFFFFFFFFF;
    END;

    SET r = a / b;
  END;

  SET OutputRoot.XML.Data.Result = r;
END;

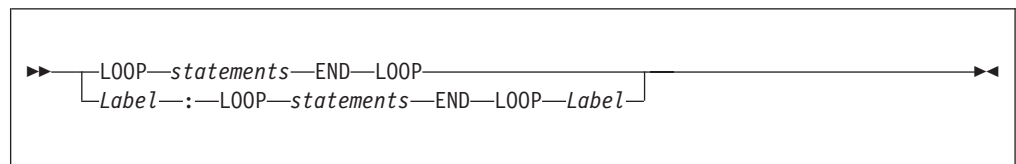
```

LOOP statement

The LOOP statement executes the sequence of statements repeatedly and unconditionally.

Ensure that the logic of the program provides some means of terminating the loop. You can use either LEAVE or RETURN statements.

Syntax



If present, *Label* gives the statement a name. This has no effect on the behavior of the LOOP statement, but allows *statements* to include ITERATE and LEAVE statements or other labelled statements, which in turn include ITERATE and LEAVE. The second *Label* can be present only if the first *Label* is present and, if it is, the labels must be identical.

Two or more labelled statements at the same level can have the same *Label* but this partly negates the advantage of the second *Label*. The advantage is that it unambiguously and accurately matches each END with its LOOP. However, a labelled statement within *statements* cannot have the same label, because this makes the behavior of the ITERATE and LEAVE statements ambiguous.

The LOOP statement is useful in cases where the required logic dictates that a loop is always exited part way through. This is because, in these cases, the testing of a loop condition that occurs in REPEAT or WHILE statements is both unnecessary and wasteful.

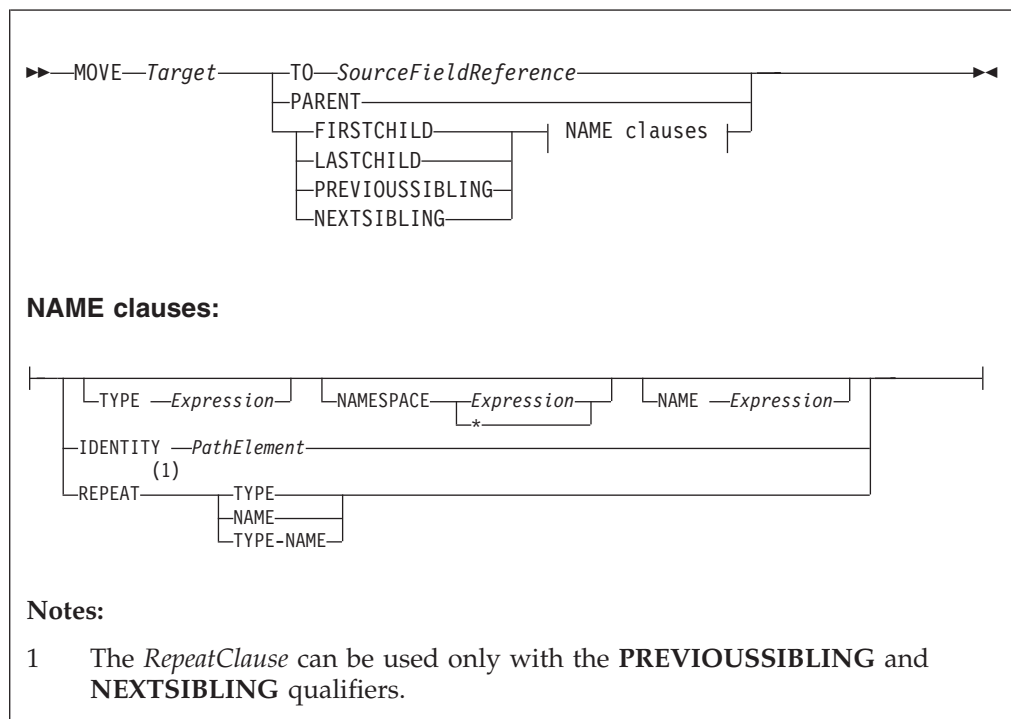
Example

```
DECLARE i INTEGER;
SET i = 1;
X : LOOP
  ...
  IF i >= 4 THEN
    LEAVE X;
  END IF;
  SET i = i + 1;
END LOOP X;
```

MOVE statement

The MOVE statement changes the field to which a reference variable identified by target points.

Syntax



If you include a TO clause, it changes the target reference to point to the same entity as that pointed to by source. This can either be a message field or a declared variable.

If you include a PARENT, PREVIOUSIBLING, NEXTSIBLING, FIRSTCHILD, or LASTCHILD clause, the MOVE statement attempts to move the target reference variable in the direction specified relative to its current position. If any field exists in the given direction, the move succeeds. If there is no such field, the move fails; that is the reference variable continues to point to the same field or variable as before, and the LASTMOVE function returns false. You can use the LASTMOVE function to determine the success or failure of a move.

If a TYPE clause, NAME clause, or both are present, the target is again moved in the direction specified (PREVIOUSIBLING or NEXTSIBLING, or FIRSTCHILD or LASTCHILD) but to a field with the given type, name, or both. This is particularly

useful when the name or type (or both) of the target field is known, because this reduces the number of MOVE statements required to navigate to a field. This is because fields that do not match the criteria are skipped over; this can also include unexpected message tree fields, for example, those representing whitespace.

If the specified move cannot be made (that is, a field with the given type or name does not exist), the target remains unchanged and the LASTMOVE function returns false. The TYPE clause, NAME clause, or both clauses can contain any expression that returns a value of a suitable data type (INTEGER for type and CHARACTER for name). An exception is thrown if the value supplied is NULL.

Two further clauses, NAMESPACE and IDENTITY enhance the functionality of the NAME clause.

The NAMESPACE clause takes any expression that returns a non-null value of type character. It also takes an * indicating any namespace. Note that this cannot be confused with an expression because * is not a unary operator in ESQL.

The meaning depends on the presence of NAME and NAMESPACE clauses as follows:

NAMESPACE	NAME	Element located by...
No	No	Type, index, or both
No	Yes	Name in the default namespace
*	Yes	Name
Yes	No	Namespace
Yes	Yes	Name and namespace

The IDENTITY clause takes a single path element in place of the TYPE, NAMESPACE, and NAME clauses and follows all the rules described in the topic for field references (see “ESQL field references” on page 792).

When using MOVE with PREVIOUSIBLING or NEXTSIBLING, you can specify REPEAT, TYPE, and NAME keywords that move the target to the previous or next field with the same type and name as the current field. The REPEAT keyword is particularly useful when moving to a sibling of the same kind, because you do not have to write expressions to define the type and name.

Example

```
MOVE cursor FIRSTCHILD TYPE 0x01000000 NAME 'Field1';
```

This example moves the reference variable cursor to the first child field of the field to which cursor is currently pointing and that has the type 0x01000000 and the name Field1.

The MOVE statement never creates new fields.

A common usage of the MOVE statement is to step from one instance of a repeating structure to the next. The fields within the structure can then be accessed by using a relative field reference. For example:

```
WHILE LASTMOVE(sourceCursor) DO
  SET targetCursor.ItemNumber = sourceCursor.item;
  SET targetCursor.Description = sourceCursor.name;
  SET targetCursor.Price      = sourceCursor.prc;
```

```

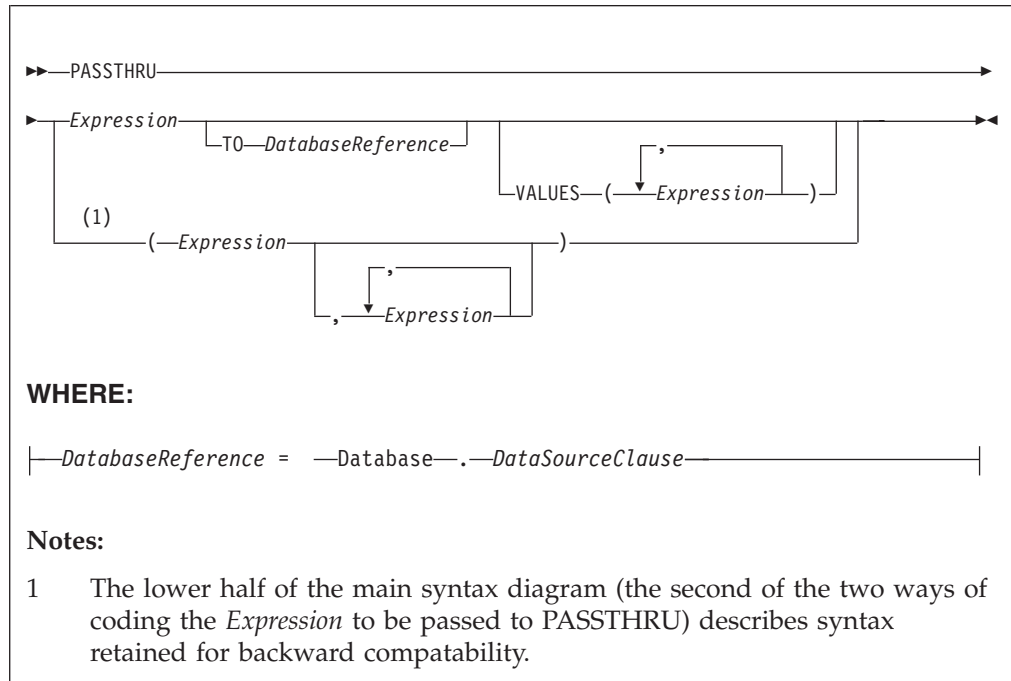
SET targetCursor.Tax      = sourceCursor.prc * 0.175;
SET targetCursor.quantity = 1;
CREATE NEXTSIBLING OF targetCursor AS targetCursor REPEAT;
MOVE sourceCursor NEXTSIBLING REPEAT TYPE NAME;
END WHILE;

```

For more information about reference variables, and an example of moving a reference variable, see “Creating dynamic field references” on page 179.

PASSTHRU statement

The PASSTHRU statement evaluates an expression and executes the resulting character string as a database statement.



Usage

The main use of the PASSTHRU statement is to issue administrative commands to databases (to, for example, create a table).

Note: You are not recommended to use PASSTHRU to call stored procedures. This is because of the limitations that PASSTHRU imposes. (You cannot use output parameters, for example.) To call stored procedures, use the CALL statement instead.

The first expression is evaluated and the resulting character string is passed to the database pointed to by *DatabaseReference* (in the TO clause) for execution. If the TO clause is not specified, the database pointed to by the node’s data source attribute is used.

Use question marks (?) in the database string to denote parameters. The parameter values are supplied by the VALUES clause.

If the VALUES clause is specified, its expressions are evaluated and passed to the database as parameters; (that is, their values are substituted for the question marks in the database statement).

If there is only one VALUE expression, the result may or may not be a list. If it is a list, the list's scalar values are substituted for the question marks, sequentially. If it is not a list, the single scalar value is substituted for the (single) question mark in the database statement. If there is more than one VALUE expression, none of the expressions should evaluate to a list. Their scalar values are substituted for the question marks, sequentially.

Because the database statement is constructed by the user program, there is no absolute need to use parameter markers (that is, the question marks) or the VALUES clause, because the whole of the database statement could be supplied, as a literal string, by the program. However, it is recommended that you use parameter markers whenever possible, because this reduces the number of different statements that need to be prepared and stored in the database and the broker.

Database reference

A database reference is a special case of the field references used to refer to message trees. It consists of the word "Database" followed by a data source name (that is, the name of a database instance).

You can specify the data source name directly or by an expression enclosed in braces ({...}). A directly-specified data source name is subject to name substitution. That is, if the name used has been declared to be a known name, the value of the declared name is used rather than the name itself (see "DECLARE statement" on page 851).

Handling errors

It is possible for errors to occur during PASSTHRU operations. For example, the database may not be operational or the statement may be invalid. In these cases, an exception is thrown (unless the node has its throw exception on database error property set to FALSE). These exceptions set appropriate SQL code, state, native error, and error text values and can be dealt with by error handlers (see the DECLARE HANDLER statement).

For further information about handling database errors, see "Capturing database state" on page 219.

Examples

The following example creates the "Customers" table in schema "Shop" in database DSN1:

```
PASSTHRU 'CREATE TABLE Shop.Customers (  
  CustomerNumber INTEGER,  
  FirstName      VARCHAR(256),  
  LastName       VARCHAR(256),  
  Street         VARCHAR(256),  
  City           VARCHAR(256),  
  Country        VARCHAR(256)  
)' TO Database.DSN1;
```

If, as in the last example, the ESQL statement is specified as a string literal, you must put single quotes around it. If, however, it is specified as a variable, omit the quotes. For example:

```
SET myVar = 'SELECT * FROM user1.stocktable';
SET OutputRoot.XML.Data[] = PASSTHRU(myVar);
```

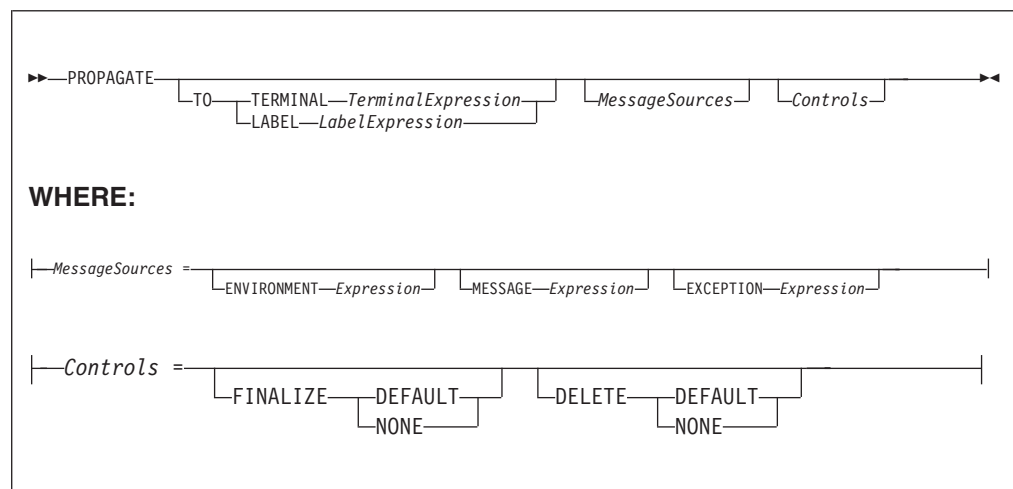
The following example “drops” (that is, deletes) the “Customers” table in schema “Shop” in database DSN1:

```
PASSTHRU 'DROP TABLE Shop.Customers' TO Database.DSN1;
```

PROPAGATE statement

The PROPAGATE statement propagates a message to the downstream nodes.

Syntax



You can use the PROPAGATE statement in Compute and Database nodes, but not in Filter nodes. The additions to this statement assist in error handling - see “Coding ESQL to handle errors” on page 214.

TO TERMINAL clause

If the TO TERMINAL clause is present, *TerminalExpression* is evaluated. If the result is of type CHARACTER, a message is propagated to a terminal according to the rule:

```
'nowhere' : no propagation
'failure' : Failure
'out'     : Out
'out 1'   : Out1
'out 2'   : Out2
'out 3'   : Out3
'out 4'   : Out4
```

Tip: Terminal names are case sensitive so, for example, “Out1” does not match any terminal.

If the result of *TerminalExpression* is of type INTEGER, a message is propagated to a terminal according to the rule:

```
-2 : no propagation
-1 : failure
0  : out
```

```
1 : out1
2 : out2
3 : out3
4 : out4
```

If the result of *TerminalExpression* is neither a CHARACTER nor an INTEGER, the broker throws an exception.

If there is neither a TO TERMINAL nor a TO LABEL clause, the broker propagates a message to the “out” terminal.

Tip: Using character values in terminal expressions leads to the most natural and readable code. Integer values, however, are easier to manipulate in loops and marginally faster.

TO LABEL clause

If the TO LABEL clause is present, *LabelExpression* is evaluated. If the result is of type CHARACTER **and** there is a Label node with a label attribute that matches *LabelExpression*, in the same flow, the broker propagates a message to that node.

Tip: Labels, like terminals, are case sensitive. Also, note that, as with route to Label nodes, it is the *labelName* attribute of the Label node that defines the target, not the node’s label itself.

If the result of *LabelExpression* is NULL or not of type CHARACTER, or there is no matching Label node in the flow, the broker throws an exception.

If there is neither a TO TERMINAL nor a TO LABEL clause, the broker propagates a message to the “out” terminal.

MessageSources clauses

The MessageSources clauses select the message trees to be propagated. This clause is only applicable to the Compute node (it has no effect in the Database node).

The values that you can specify in MessageSources clauses are:

```
ENVIRONMENT :
  InputLocalEnvironment
  OutputLocalEnvironment
```

```
Message :
  InputRoot
  OutputRoot
```

```
ExceptionList :
  InputExceptionList
  OutputExceptionList
```

If there is no MessageSources clause, the node’s “*compute mode*” attribute is used to determine which messages are propagated.

FINALIZE clause

Finalization is a process that fixes header chains and makes the Properties folder match the headers. If present, the FINALIZE clause allows finalization to be controlled.

This clause is only applicable to the Compute node (it has no effect in a Database node).

If FINALIZE is set to DEFAULT, or the FINALIZE clause is absent, the output message (but not the Environment, Local Environment or Exception List) is finalized before propagation.

If FINALIZE is set to NONE, no finalization takes place.

DELETE clause

The DELETE clause allows the clearing of the output local environment, message, and exception list to be controlled.

The DELETE clause is only applicable to the Compute node (it has no effect in a Database node).

If DELETE is set to DEFAULT, or the DELETE clause is absent, the output local environment, message, and exception list are all cleared and their memory recovered immediately after propagation.

If DELETE is set to NONE, nothing is cleared.

Note that it is the output trees that are finalized are cleared, regardless of which ones are propagated.

The Compute node allows its output message to be changed by other nodes (by the other nodes changing their input message). However, a message created by a Compute node cannot be changed by another node after:

- It has been finalized
- It has reached any output or other node which generates a bit-stream

Propagation is a synchronous process. That is, the next statement is not executed until all the processing of the message in downstream nodes has completed. Be aware that this processing might throw exceptions and that, if these exceptions are not caught, they will prevent the statement following the PROPAGATE call being reached. This may be what the logic of your flow requires but, if it is not, you can use a handler to catch the exception and perform the necessary actions. Note that exceptions thrown downstream of a propagate, if not caught, will also prevent the final automatic actions of a Compute or Database node (for example, issuing a COMMIT Transaction set to Commit) from taking place.

```
DECLARE i INTEGER 1;
DECLARE count INTEGER;
SET count = CARDINALITY(InputRoot.XML.Invoice.Purchases."Item"[])

WHILE i <= count DO
  --use the default tooling-generated procedure for copying message headers
  CALL CopyMessageHeaders();
  SET OutputRoot.XML.BookSold.Item = InputRoot.XML.Invoice.Purchases.Item[i];
  PROPAGATE;
  SET i = i+1;
END WHILE;
RETURN FALSE;
```

Here are the messages produced on the OUT terminal by the PROPAGATE statement:

```
<BookSold>
<Item>
  <Title Category="Computer" Form="Paperback" Edition="2">The XML Companion </Title>
  <ISBN>0201674866</ISBN>
  <Author>Neil Bradley</Author>
  <Publisher>Addison-Wesley</Publisher>
  <PublishDate>October 1999</PublishDate>
```

```

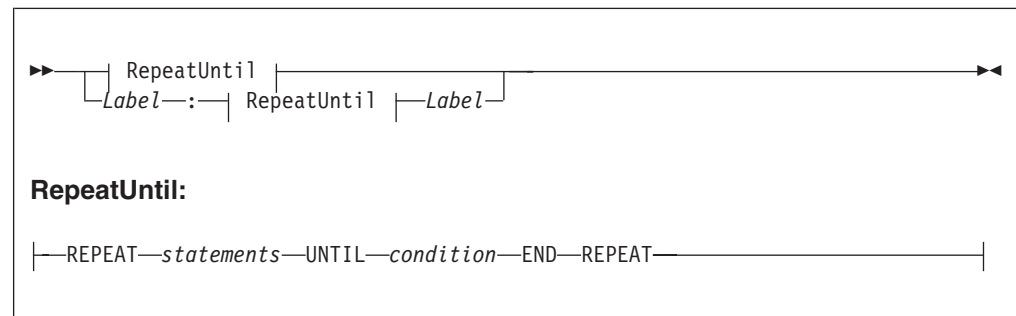
    <UnitPrice>27.95</UnitPrice>
    <Quantity>2</Quantity>
  </Item>
</BookSold>
<BookSold>
  <Item>
    <Title Category="Computer" Form="Paperback" Edition="2">A Complete Guide to
      DB2 Universal Database</Title>
    <ISBN>1558604820</ISBN>
    <Author>Don Chamberlin</Author>
    <Publisher>Morgan Kaufmann Publishers</Publisher>
    <PublishDate>April 1998</PublishDate>
    <UnitPrice>42.95</UnitPrice>
    <Quantity>1</Quantity>
  </Item>
</BookSold>
<BookSold>
  <Item>
    <Title Category="Computer" Form="Hardcover" Edition="0">JAVA 2 Developers
      Handbook</Title>
    <ISBN>0782121799</ISBN>
    <Author>Phillip Heller, Simon Roberts </Author>
    <Publisher>Sybex, Inc.</Publisher>
    <PublishDate>September 1998</PublishDate> <UnitPrice>59.99</UnitPrice>
    <Quantity>1</Quantity>
  </Item>
</BookSold>

```

REPEAT statement

The REPEAT statement processes a sequence of statements and then evaluates the condition expression.

Syntax



The REPEAT statement repeats the steps until condition is TRUE. Ensure that the logic of the program is such that the loop terminates. If the condition evaluates to UNKNOWN, the loop does **not** terminate.

If present, the *Label* gives the statement a name. This has no effect on the behavior of the REPEAT statement, but allows statements to include ITERATE and LEAVE statements or other labelled statements, which in turn include ITERATE and LEAVE. The second *Label* can be present only if the first *Label* is present and, if it is, the labels must be identical. Two or more labelled statements at the same level can have the same label, but this partly negates the advantage of the second *Label*. The advantage is that it unambiguously and accurately matches each END with its

REPEAT. However, a labelled statement within statements cannot have the same label because this makes the behavior of the ITERATE and LEAVE statements ambiguous.

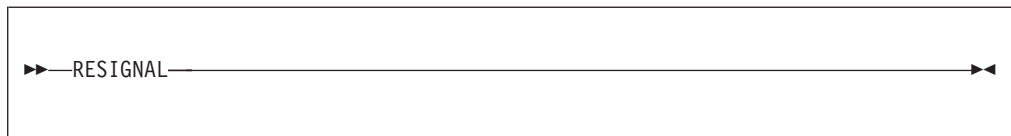
Example

```
DECLARE i INTEGER;
SET i = 1;
X : REPEAT
  ...
  SET i = i + 1;
UNTIL
  i >= 3
END REPEAT X;
```

RESIGNAL statement

The RESIGNAL statement re-throws the current exception (if there is one).

Syntax



RESIGNAL re-throws the current exception (if there is one). You can use it only in error handlers..

Typically, RESIGNAL is used when an error handler catches an exception that it can't handle. The handler uses RESIGNAL to re-throw the original exception so that a handler in higher-level scope has the opportunity to handle it.

Because the handler throws the original exception, rather than a new (and therefore different) one:

1. The higher-level handler is not affected by the presence of the lower-level handler.
2. If there is no higher-level handler, you get a full error report in the event log.

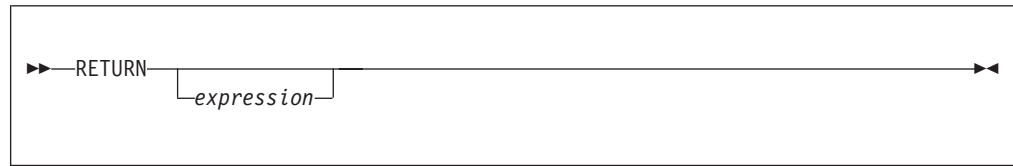
Example

```
RESIGNAL;
```

RETURN statement

The RETURN statement ends processing. What happens next depends on the programming context in which the RETURN statement is issued.

Syntax



When used in a function, the RETURN statement stops processing of that function and returns control to the calling expression. The *expression* (which must be present) is evaluated and acts as the return value of the function. It is an error for a function to return by running off the list of statements. The data type of the returned value must be the same as that in the function's declaration.

When used in a procedure, the RETURN statement stops processing of that procedure and returns control to the calling CALL statement. A RETURN statement used within a procedure must not have an *expression*.

When used in a Filter, Compute, or Database node's mainline code, the RETURN statement stops processing of the node's ESQL and passes control to the next node. In these cases, if *expression* is present, it must evaluate to a BOOLEAN value. If *expression* is not present, a Filter node assumes a value of UNKNOWN and propagates to its unknown terminal; Compute and Database nodes propagate to their out terminals.

The following table describes the differences between the RETURN statement when used in the Compute, Filter, and Database nodes.

	Return value	Result
Compute node:		
RETURN	TRUE	Propagate message to out terminal.
	FALSE	Do not propagate.
	UNKNOWN	Do not propagate.
RETURN;		Propagate message to out terminal.
Filter node:		
RETURN	TRUE	Propagate message to true terminal.
	FALSE	Propagate message to false terminal.
	UNKNOWN	Propagate message to unknown terminal.
RETURN;		Propagate message to unknown terminal.
Database node:		
RETURN	TRUE	Propagate message to out terminal.
	FALSE	Do not propagate.
	UNKNOWN	Do not propagate.

RETURN;	Propagate message to out terminal.
---------	------------------------------------

Example

The following example, which is based on “Example message” on page 991, illustrates how this statement can be used:

```
-- Declare variables --
DECLARE a INT;
DECLARE PriceTotal FLOAT;
DECLARE NumItems INT;

-- Initialize values --
SET a = 1;
SET NumItems = 0;
SET PriceTotal = 0.0;

-- Calculate value of order, however if this is a bulk purchase, the --
-- order will need to be handled differently (discount given) so return TRUE --
-- or FALSE depending on the size of the order --
WHILE a <= CARDINALITY(Invoice.Purchases.Item[a]) DO
    SET NumItems = NumItems + Invoice.Purchases.Item[a].Quantity;
    SET PriceTotal = PriceTotal + Invoice.Purchases.Item[a].UnitPrice;
    SET a = a + 1;
END;
RETURN PriceTotal/NumItems > 42;
```

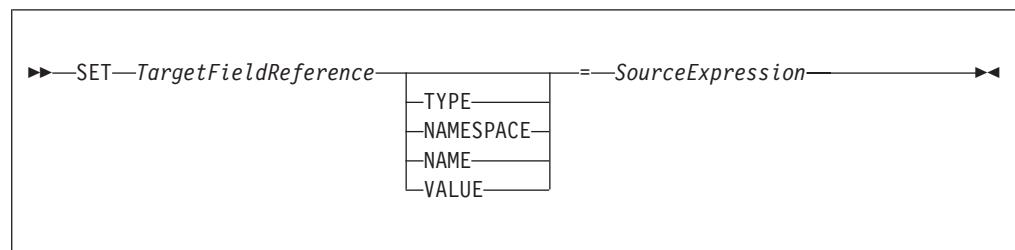
If the average price of items is greater than 42, TRUE is returned; otherwise FALSE is returned. Thus, a Filter node could route messages describing expensive items down a different path from messages describing inexpensive items.

See “PROPAGATE statement” on page 874 for an example of RETURN FALSE to prevent the implicit propagate at the end of processing in a Compute node.

SET statement

The SET statement assigns a value to a variable.

Syntax



Introduction

TargetFieldReference identifies the target of the assignment. The target can be any of the following:

- A declared scalar variable
- A declared row variable
- One of the predefined row variables (for example, *InputRoot*)
- A field within any kind of row variable (that is, a sub tree or conceptual row)

- A list of fields within any kind of row variable (that is, a conceptual list)
- A declared reference variable that points to any of the above

The target cannot be any kind of database entity.

SourceExpression is an expression which supplies the value to be assigned. It may be any kind of expression and may return a scalar, row or list value.

Assignment to scalar variables

If the target is a declared scalar variable, *SourceExpression* is evaluated and assigned to the variable. If need be, its value is converted to the data type of the variable. If this conversion is not possible, there will be either an error at deploy time or an exception at run time.

Null values are handled in exactly the same way as any other value. That is, if the expression evaluates to null, the value “null” is assigned to the variable.

For scalar variables the TYPE, NAME, NAMESPACE, and VALUE clauses are meaningless and are not allowed.

Assignment to rows, lists, and fields

If the target is a declared row variable, one of the predefined row variables, a field within any kind of row variable, a list of fields within any kind of row variable, or a declared reference variable that points to any of these things, the ultimate target is a field. In these cases, the target field is navigated to (creating the fields if necessary).

If array indices are used in *TargetFieldReference*, the navigation to the target field can only create fields on the direct path from the root to the target field. For example, the following SET statement requires that at least one instance of Structure already exists in the message:

```
SET OutputRoot.XML.Message.Structure[2].Field = ...
```

The target field’s value is set according to a set of rules, based on:

1. The presence or absence of the TYPE, NAME, NAMESPACE, or VALUE clauses
 2. The data type returned by the source expression
1. If no TYPE, NAME, NAMESPACE, or VALUE clause is present (which is the most common case) the outcome depends on whether *SourceExpression* evaluates to a scalar, a row, or a list:
 - If *SourceExpression* evaluates to a scalar, the value of the target field is set to the value returned by *SourceExpression*, except that, if the result is null, the target field is discarded. Note that the new value of the field may not be of the same data type as its previous value.
 - If *SourceExpression* evaluates to a row:
 - a. The target field is identified.
 - b. The target field’s value is set.
 - c. The target field’s child fields are replaced by a new set, dictated by the structure and content of the list.
 - If *SourceExpression* evaluates to a list:
 - a. The set of target fields in the target tree are identified.
 - b. If there are too few target fields, more are created; if there are too many, the extra ones are removed.
 - c. The target fields’ values are set.

- d. The target fields' child fields are replaced by a new set, dictated by the structure and content of the list.

For further information on working with elements of type `list` see "Working with elements of type `xsd:: list`"

2. If a `TYPE` clause is present, the type of the target field is set to the value returned by *SourceExpression*. An exception is thrown if the returned value is not scalar, is not of type `INTEGER`, or is `NULL`.
3. If a `NAMESPACE` clause is present, the namespace of the target field is set to the value returned by *SourceExpression*. An exception is thrown if the returned value is not scalar, is not of type `CHARACTER`, or is `NULL`.
4. If a `NAME` clause is present, the name of the target field is set to the value returned by *SourceExpression*. An exception is thrown if the returned value is not scalar, is not of type `CHARACTER`, or is `NULL`.
5. If a `VALUE` clause is present, the value of the target field is changed to that returned by *SourceExpression*. An exception is thrown if the returned value is not scalar.

Notes

`SET` statements are particularly useful in `Compute` nodes that modify a message, either changing a field or adding a new field to the original message. `SET` statements are also useful in `Filter` and `Database` nodes, to set declared variables or the fields in the `Environment` tree or `Local Environment` trees. You can use statements such as the following in a `Compute` node that modifies a message:

```
SET OutputRoot = InputRoot;  
SET OutputRoot.XML.Order.Name = UPPER(InputRoot.XML.Order.Name);
```

This example puts one field in the message into uppercase. The first statement constructs an output message that is a complete copy of the input message. The second statement sets the value of the `Order.Name` field to a new value, as defined by the expression on the right.

If the `Order.Name` field does not exist in the original input message, it does not exist in the output message generated by the first statement. The expression on the right of the second statement returns `NULL` (because the field referenced inside the `UPPER` function call does not exist). Assigning the `NULL` value to a field has the effect of deleting it if it already exists, and so the effect is that the second statement has no effect.

If you want to assign a `NULL` value to a field without deleting the field, use a statement like this:

```
SET OutputRoot.XML.Order.Name VALUE = NULL;
```

Working with elements of type `xsd:: list`

The XML Schema specification permits an element or attribute to contain a list of values based on a simple type with the individual values separated by white space.

Consider the following XML input message:

```
<message1>  
  <listE1 listAttr="one two three"> four five six</listE1>  
</message1>
```

In the resulting message tree, an `xsd:list` type is represented as a name node with an anonymous value child for each list item. This allows repeating lists to be handled without any loss of information.

Repeating lists appear as sibling name elements, each of which has its own anonymous value child nodes for its respective list items. The preceding example message produces the following logical tree:

```
MRM
  listEl (Name)
    listAttr (Name)
      "one" (Value)
      "two" (Value)
      "three" (Value)
      "four" (Value)
      "five" (Value)
      "six" (Value)
```

Individual list items can be accessed as `ElementName.*[n]`. For example:

```
SET OutputRoot.MRM.listEl.listAttr.*[3] = ...
```

modifies the third item of `listAttr`.

Mapping between a list and a repeating element

Consider the form of the following XML input message:

```
<MRM>
  <inner>abcde fghij 12345</inner>
</MRM>
```

where the element `inner` is of type `xsd:list`, so it has three associated string values, rather than a single value.

If you want to copy the three values into an output message, where each value is associated with an instance of repeating elements as follows:

```
<MRM>
  <str1>abcde</str1>
  <str1>fghij</str1>
  <str1>12345</str1>
</MRM>
```

it is reasonable to assume that the following ESQL syntax works:

```
DECLARE D INTEGER;
SET D = CARDINALITY(InputBody.str1.*[]);
DECLARE M INTEGER 1;
WHILE M <= D DO
  SET OutputRoot.MRM.str1[M] = InputBody.inner.*[M];
  SET M = M + 1;
END WHILE;
```

However, the statement:

```
SET OutputRoot.MRM.str1[M] = InputBody.inner.*[M];
```

requests a tree copy from source to target. Since the target element does not yet exist, it is created and its value *and type* are set from the source.

This is consistent with ESQL's behavior elsewhere, but in the case of elements having values of type `list`, this code can produce spurious validation errors.

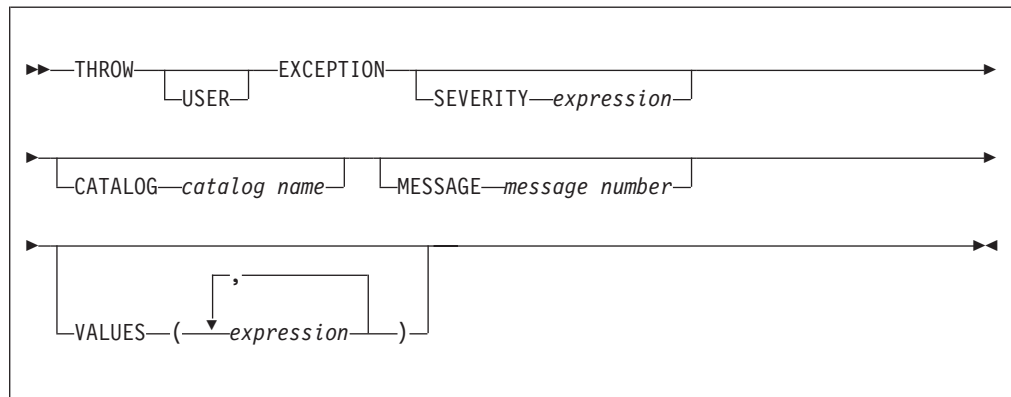
To avoid this problem, you are recommended to use the “FIELDVALUE function” on page 933 to explicitly retrieve only the value of the source element, as follows:

```
SET OutputRoot.MRM.str1[M] = FIELDVALUE(InputBody.inner.*[M]);
```

THROW statement

The THROW statement generates a user exception.

Syntax



The USER keyword indicates the type of exception being thrown. (Currently, only USER exceptions are supported, and if you omit the USER keyword the exception defaults to a USER exception anyway.) Although, at present, specifying the USER keyword has no effect, you are nevertheless recommended to include it, because:

- If future broker releases support other types of exception, and the default type changes, your code will not need to be changed.
- It makes it clear that this is a user exception.

SEVERITY is an optional clause that determines the severity associated with the exception. The clause can contain any expression that returns a non-NULL, integer value. If you omit the clause, it defaults to 1.

CATALOG is an optional clause; if you omit it, it defaults to the WebSphere Message Broker current version catalog. To use the current WebSphere Message Broker version message catalog explicitly, use BIPV600 on all operating systems.

MESSAGE is an optional clause; if you omit it, it defaults to the first message number of the block of messages provided for using THROW statements in WebSphere Message Broker catalog (2951). If you enter a message number in the THROW statement, you can use message numbers 2951 to 2999. Alternatively, you can generate your own catalog by following the instructions in Using event logging from a user-defined extension.

Use the optional VALUES field to insert data into your message. You can insert any number of pieces of information, but the messages supplied (2951 - 2999) cater for eight inserts only.

Examples

Here are some examples of how you might use a THROW statement:

-

```

    THROW USER EXCEPTION;
•
    THROW USER EXCEPTION CATALOG 'BIPv600' MESSAGE 2951 VALUES(1,2,3,4,5,6,7,8) ;
•
    THROW USER EXCEPTION CATALOG 'BIPv600' MESSAGE 2951 VALUES('The SQL State: ',
        SQLSTATE, 'The SQL Code: ', SQLCODE, 'The SQLNATIVEERROR: ', SQLNATIVEERROR,
        'The SQL Error Text: ', SQLERRORTXT ) ;
•
    THROW USER EXCEPTION CATALOG 'BIPv600' MESSAGE 2951 ;
•
    THROW USER EXCEPTION CATALOG 'MyCatalog' MESSAGE 2951 VALUES('Hello World') ;
•
    THROW USER EXCEPTION MESSAGE 2951 VALUES('Insert text 1', 'Insert text 2') ;

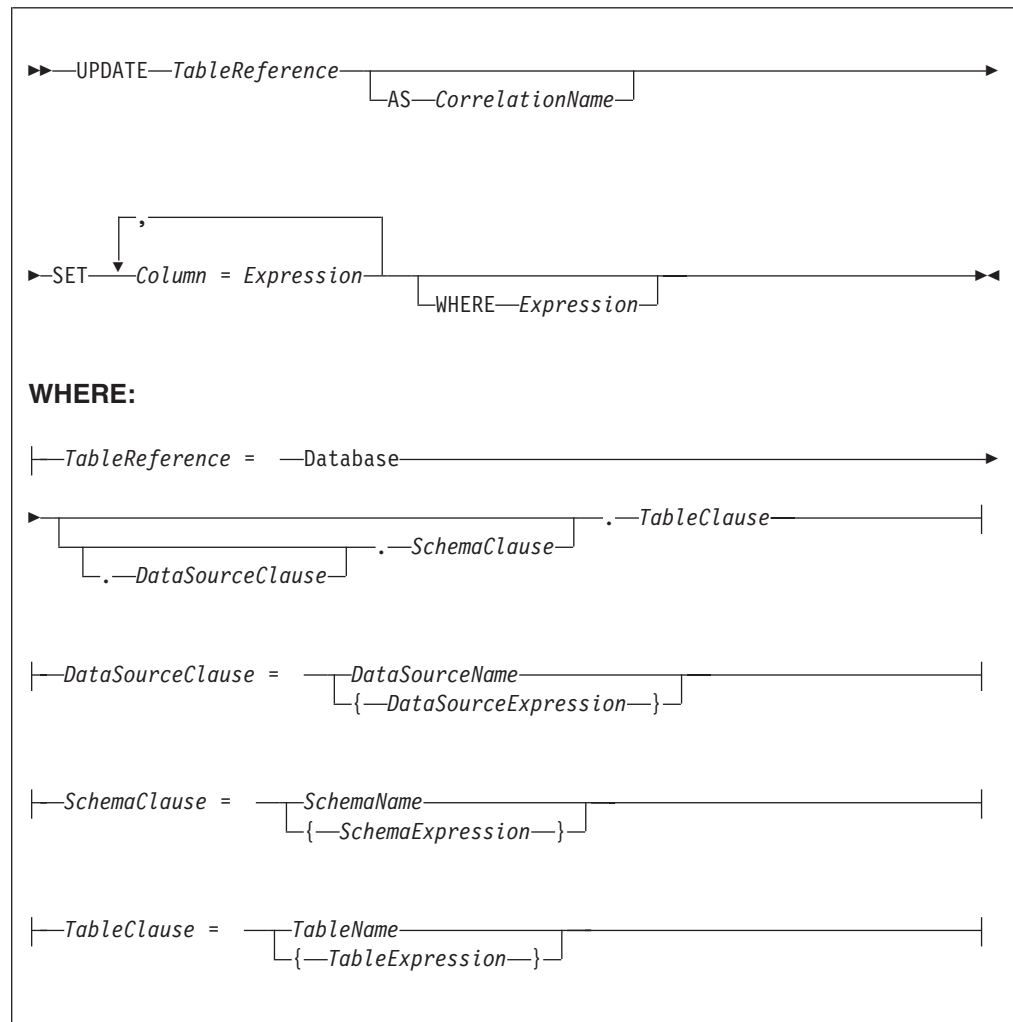
```

For more information about how to throw an exception, and details of SQLSTATE, SQLCODE, SQLNATIVEERROR, and SQLERRORTXT, see “ESQL database state functions” on page 892.

UPDATE statement

The UPDATE statement changes the values of specified columns, in selected rows, in a table in an external database.

Syntax



All rows for which the WHERE clause expression evaluates to TRUE are updated in the table identified by *TableReference*. Each row is examined in turn and a variable is set to point to the current row. Typically, the WHERE clause expression uses this variable to access column values and thus cause rows to be updated, or retained unchanged, according to their contents. The variable is referred to by *CorrelationName* or, in the absence of an AS clause, by *TableName*. When a row has been selected for updating, each column named in the SET clause is given a new value as determined by the corresponding expression. These expressions can, if you wish, refer to the current row variable.

Table reference

A table reference is a special case of the field references used to refer to message trees. It always starts with the word "Database" and may contain any of the following:

- A table name only
- A schema name and a table name
- A data source name (that is, the name of a database instance), a schema name, and a table name

In each case, the name may be specified directly or by an expression enclosed in braces ({...}). A directly-specified data source, schema, or table name is subject to name substitution. That is, if the name used has been declared to be a known name, the value of the declared name is used rather than the name itself (see “DECLARE statement” on page 851).

If a schema name is not specified, the default schema for the broker’s database user is used.

If a data source name is not specified, the database pointed to by the node’s data source attribute is used.

The WHERE clause

The WHERE clause expression can use any of the broker’s operators and functions in any combination. It can refer to table columns, message fields, and any declared variables or constants.

However, be aware that the broker treats the WHERE clause expression by examining the expression and deciding whether the whole expression can be evaluated by the database. If it can, it is given to the database. In order to be evaluated by the database, it must use only those functions and operators supported by the database.

The WHERE clause can, however, refer to message fields, correlation names declared by containing SELECTs, and to any other declared variables or constants within scope.

If the whole expression cannot be evaluated by the database, the broker looks for top-level AND operators and examines each sub-expression separately. It then attempts to give the database those sub-expressions that it can evaluate, leaving the broker to evaluate the rest. You need to be aware of this situation for two reasons:

1. Apparently trivial changes to WHERE clause expressions can have large effects on performance. You can determine how much of the expression was given to the database by examining a user trace.
2. Some databases’ functions exhibit subtle differences of behavior from those of the broker.

Handling errors

It is possible for errors to occur during update operations. For example, the database may not be operational, or the table may have constraints defined that the new values would violate. In these cases, an exception is thrown (unless the node has its `throw exception on database error` property set to `FALSE`). These exceptions set appropriate SQL code, state, native error, and error text values and can be dealt with by error handlers (see the `DECLARE HANDLER` statement).

For further information about handling database errors, see “Capturing database state” on page 219.

Examples

The following example assumes that the `dataSource` property of the Database node has been configured, and that the database it identifies has a table called

STOCKPRICES, with columns called COMPANY and PRICES. It updates the PRICE column of the rows in the STOCKPRICES table whose COMPANY column matches the value given in the Company field in the message.

```
UPDATE Database.StockPrices AS SP
  SET PRICE = InputBody.Message.StockPrice
  WHERE SP.COMPANY = InputBody.Message.Company
```

In the following example (which make similar assumptions), the SET clause expression refers to the existing value of a column and thus decrements the value by an amount in the message:

```
UPDATE Database.INVENTORY AS INV
  SET QUANTITY = INV.QUANTITY - InputBody.Message.QuantitySold
  WHERE INV.ITEMNUMBER = InputBody.Message.ItemNumber
```

The following example updates multiple columns:

```
UPDATE Database.table AS T
  SET column1 = T.column1+1,
      column2 = T.column2+2;
```

Note that the column names (on the left of the "=") are single identifiers. They must not be qualified with a table name or correlation name. In contrast, the references to database columns in the expressions (to the right of the "=") must be qualified with the correlation name.

The next example shows the use of calculated data source, schema, and table names:

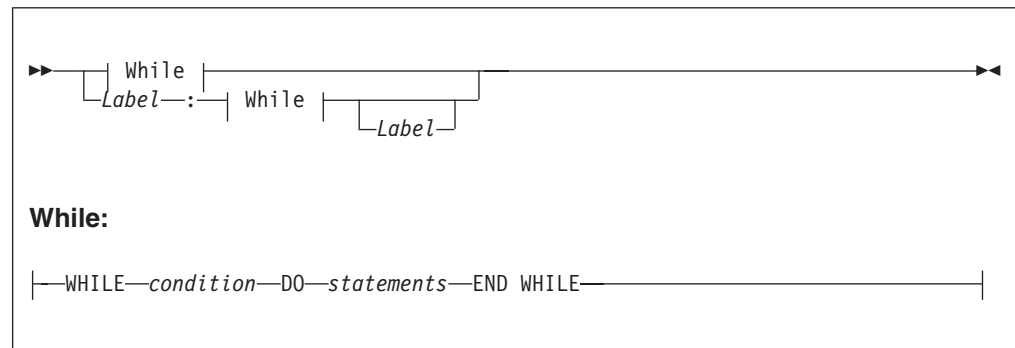
```
-- Declare variables to hold the data source, schema and table names
-- and set their default values
DECLARE Source CHARACTER 'Production';
DECLARE Schema CHARACTER 'db2admin';
DECLARE Table CHARACTER 'DynamicTable1';
-- Code which calculates their actual values comes here

-- Update rows in the table
UPDATE Database.{Source}.{Schema}.{Table} AS R SET Value = 0;
```

WHILE statement

The WHILE statement evaluates a condition expression, and if it is TRUE executes a sequence of statements.

Syntax



The WHILE statement repeats the steps specified in DO as long as *condition* is TRUE. It is your responsibility to ensure that the logic of the program is such that the loop terminates. If *condition* evaluates to UNKNOWN, the loop terminates immediately.

If present, *Label* gives the statement a name. This has no effect on the behavior of the WHILE statement itself, but allows statements to include ITERATE and LEAVE statements or other labelled statements, which in turn include them. The second *Label* can be present only if the first *Label* is present and if it is, the labels must be identical. It is not an error for two or more labelled statements at the same level to have the same *Label*, but this partly negates the advantage of the second *Label*. The advantage is that it unambiguously and accurately matches each END with its WHILE. However, it is an error for a labelled statement within statements to have the same label, because this makes the behavior of the ITERATE and LEAVE statements ambiguous.

Example

For example:

```
DECLARE i INTEGER;
SET i = 1;
X : WHILE i <= 3 DO
...
SET i = i + 1;
END WHILE X;
```

ESQL functions: reference material, organized by function type

The following table summarizes the functions available in ESQL, and what they do.

CATEGORY	FUNCTIONS	RELATED KEYWORDS
Variable manipulation		
Manipulation of all sources of variables		
Basic manipulation of all types of variable	<ul style="list-style-type: none"> “CAST function” on page 940 	<ul style="list-style-type: none"> ENCODING, CCSID, AS
Selective assignment to any variable	<ul style="list-style-type: none"> “CASE function” on page 939 “COALESCE function” on page 979 	<ul style="list-style-type: none"> ELSE, WHEN, THEN, END
Creation of values	<ul style="list-style-type: none"> “UIDASBLOB function” on page 982 “UIDASCHAR function” on page 983 	-

Manipulation of message trees		
Assignment to and deletion from a message tree	<ul style="list-style-type: none"> • “SELECT function” on page 954 (used with SET statement) • “ROW constructor function” on page 960 • “LIST constructor function” on page 961 	<ul style="list-style-type: none"> • FROM, AS, ITEM, THE, SUM, COUNT, MAX, MIN
Information relating to message trees or subtrees	<ul style="list-style-type: none"> • “ASBITSTREAM function” on page 925 • “BITSTREAM function (deprecated)” on page 929 • “FIELDNAME function” on page 929 • “FIELDNAMESPACE function” on page 930 • “FIELDTYPE function” on page 930 	-
Processing Lists	<ul style="list-style-type: none"> • CARDINALITY, see “CARDINALITY function” on page 936 for details. • EXISTS, see “EXISTS function” on page 937 for details. • SINGULAR, see “SINGULAR function” on page 937 for details. • THE, see “THE function” on page 938 for details. 	
Processing repeating fields	<ul style="list-style-type: none"> • FOR • “SELECT function” on page 954 	<ul style="list-style-type: none"> • ALL, ANY, SOME • FROM, AS, ITEM, THE, SUM, COUNT, MAX, MIN
Processing based on data type		
String processing		
Numeric information about strings	<ul style="list-style-type: none"> • “LENGTH function” on page 916 • “POSITION function” on page 919 	IN
String conversion	<ul style="list-style-type: none"> • “UPPER and UCASE functions” on page 925 • “LOWER and LCASE functions” on page 917 	-
String manipulation	<ul style="list-style-type: none"> • “LEFT function” on page 916 • “LTRIM function” on page 917 • “OVERLAY function” on page 918 • “REPLACE function” on page 920 • “REPLICATE function” on page 920 • “RIGHT function” on page 921 • “RTRIM function” on page 921 • “SPACE function” on page 922 • “SUBSTRING function” on page 922 • “TRANSLATE function” on page 923 • “TRIM function” on page 924 	<ul style="list-style-type: none"> • LEADING, TRAILING, BOTH, FROM • PLACING, FROM, FOR • FROM FOR
Numeric processing		

Bitwise operations	<ul style="list-style-type: none"> • “BITAND function” on page 905 • “BITNOT function” on page 905 • “BITOR function” on page 906 • “BITXOR function” on page 906 	-
General	<ul style="list-style-type: none"> • “ABS and ABSVAL functions” on page 903 • “ACOS function” on page 904 • “ASIN function” on page 904 • “ATAN function” on page 904 • “ATAN2 function” on page 904 • “COS function” on page 907 • “COSH function” on page 908 • “COT function” on page 908 • “DEGREES function” on page 908 • “EXP function” on page 908 • “FLOOR function” on page 909 • “LN and LOG functions” on page 909 • “LOG10 function” on page 910 • “MOD function” on page 910 • “POWER function” on page 911 • “RADIANS function” on page 911 • “RAND function” on page 911 • “ROUND function” on page 912 • “SIGN function” on page 912 • “SIN function” on page 913 • “SINH function” on page 913 • “SQRT function” on page 913 • “TAN function” on page 914 • “TANH function” on page 914 • “TRUNCATE function” on page 915 	-
Date time processing		
	<ul style="list-style-type: none"> • “CURRENT_DATE function” on page 900 • “CURRENT_GMTDATE function” on page 901 • “CURRENT_GMTTIME function” on page 901 • “CURRENT_TIME function” on page 900 • “CURRENT_TIMESTAMP function” on page 900 • “CURRENT_GMTTIMESTAMP function” on page 901 • “LOCAL_TIMEZONE function” on page 902 • “EXTRACT function” on page 898 	YEAR, MONTH, DAY, HOUR, MINUTE, SECOND
Boolean evaluation for conditional statements		

Functions that return a boolean value	<ul style="list-style-type: none"> • BETWEEN, see “ESQL simple comparison operators” on page 798 for details. • EXISTS, see “EXISTS function” on page 937 for details. • IN, see “ESQL simple comparison operators” on page 798 for details. • LIKE, see “ESQL simple comparison operators” on page 798 for details. • “NULLIF function” on page 980 • “LASTMOVE function” on page 935 • “SAMEFIELD function” on page 935 • SINGULAR, see “SINGULAR function” on page 937 for details. 	SYMMETRIC, ASYMMETRIC, AND
Broker database interaction		
Actions on tables	<ul style="list-style-type: none"> • “PASSTHRU function” on page 980 • “SELECT function” on page 954 	<ul style="list-style-type: none"> • FROM, AS, ITEM, THE, SUM, COUNT, MAX, MIN
Results of actions	<ul style="list-style-type: none"> • “SQLCODE function” • “SQLERRORTTEXT function” on page 893 • “SQLNATIVEERROR function” on page 894 • “SQLSTATE function” on page 894 	-

Calling ESQL functions

Most ESQL functions belong to a schema called SQL and this is particularly useful if you have functions with the same name. For example, if you have created a function called SQRT, you can code:

```
/* call my SQRT function */
SET Variable1=SQRT (4);

/* call the SQL supplied function */
SET Variable2=SQL.SQRT (144);
```

Most of the functions described in this section impose restrictions on the data types of the arguments that can be passed to the function. If the values passed to the functions do not match the required data types, errors are generated at node configuration time whenever possible. Otherwise runtime errors are generated when the function is evaluated.

ESQL database state functions

ESQL provides four functions to return database state. These are:

- “SQLCODE function”
- “SQLERRORTTEXT function” on page 893
- “SQLNATIVEERROR function” on page 894
- “SQLSTATE function” on page 894

SQLCODE function

SQLCODE is a database state function that returns an INTEGER data type with a default value of 0 (zero).

Syntax



```
▶▶—SQLCODE—————▶▶
```

Within a message flow, you can access and update an external database resource using the available ESQL database functions in the Filter, Database, and Compute nodes. When making calls to an external database, you might get errors, such as a table does not exist, a database is not available, or an insert for a key that already exists.

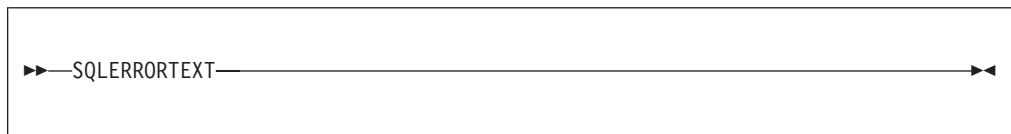
When these errors occur, the default action of the broker is to generate an exception. This behavior is determined by how you have set the property *Throw exception on database error*. If this check box is selected, the broker stops processing the node, propagates the message to the node's failure terminal, and writes the details of the error to the ExceptionList. If you want to override the default behavior and handle a database error in the ESQL in the node, clear the *Throw exception on database error* check box. The broker does not throw an exception and you must include the THROW statement to throw an exception if a certain SQL state code is not expected. See "THROW statement" on page 884 for a description of THROW.

If you choose to handle database errors in a node, you can use the database state function SQLCODE to receive information about the status of the DBMS call made in ESQL. You can include it in conditional statements in current node's ESQL to recognize and handle possible errors.

SQLERRORTEXT function

SQLERRORTEXT is a database state function that returns a CHARACTER data type with a default value of '' (empty string).

Syntax



```
▶▶—SQLERRORTEXT—————▶▶
```

Within a message flow, you can access and update an external database resource using the available ESQL database functions in the Filter, Database, and Compute nodes. When making calls to an external database, you might get errors, such as a table does not exist, a database is not available, or an insert for a key that already exists.

When these errors occur, the default action of the broker is to generate an exception. This behavior is determined by how you have set the property *Throw exception on database error*. If you have selected this check box, the broker stops processing the node, propagates the message to the node's failure terminal, and writes the details of the error to the ExceptionList. If you want to override the default behavior and handle a database error in the ESQL in the node, clear the *Throw exception on database error* check box. The broker does not throw an exception

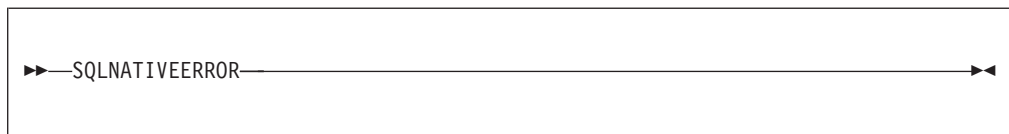
and you must include the `THROW` statement to throw an exception if a certain SQL state code is not expected. See “`THROW` statement” on page 884 for a description of `THROW`.

If you choose to handle database errors in a node, you can use the database state function `SQLERRORTEXT` to receive information about the status of the DBMS call made in ESQL. You can include it in conditional statements in current node’s ESQL to recognize and handle possible errors.

SQLNATIVEERROR function

`SQLNATIVEERROR` is a database state function that returns an `INTEGER` data type with a default value of 0 (zero).

Syntax



Within a message flow, you can access and update an external database resource using the available ESQL database functions in the Filter, Database, and Compute nodes. When making calls to an external database, you might get errors, such as a table does not exist, a database is not available, or an insert for a key that already exists.

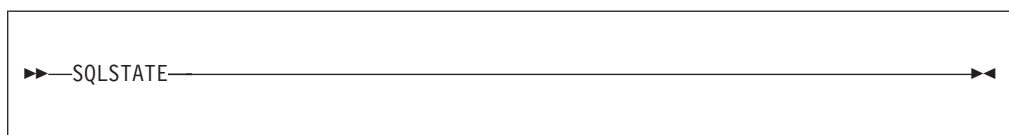
When these errors occur, the default action of the broker is to generate an exception. This behavior is determined by how you have set the property *Throw exception on database error*. If you have selected this check box, the broker stops processing the node, propagates the message to the node’s failure terminal, and writes the details of the error to the `ExceptionList`. If you want to override the default behavior and handle a database error in the ESQL in the node, clear the *Throw exception on database error* check box. The broker does not throw an exception and you must include the `THROW` statement to throw an exception if a certain SQL state code is not expected. See “`THROW` statement” on page 884 for a description of `THROW`.

If you choose to handle database errors in a node, you can use the database state function `SQLNATIVEERROR` to receive information about the status of the DBMS call made in ESQL. You can include it in conditional statements in current node’s ESQL to recognize and handle possible errors.

SQLSTATE function

`SQLSTATE` is a database state function that returns a 5 character data type of `CHARACTER` with a default value of ‘00000’ (five zeros as a string).

Syntax



Within a message flow, you can access and update an external database resource using the available ESQL database functions in the Filter, Database, and Compute

nodes. When making calls to an external database, you might get errors, such as a table does not exist, a database is not available, or an insert for a key that already exists.

When these errors occur, the default action of the broker is to generate an exception. This behavior is determined by how you have set the property *Throw exception on database error*. If you select this check box, the broker stops processing the node, propagates the message to the node's failure terminal, and writes the details of the error to the ExceptionList. If you want to override the default behavior and handle a database error in the ESQL in the node, clear the *Throw exception on database error* check box. The broker does not throw an exception and you must include the THROW statement to throw an exception if a certain SQL state code is not expected. See "THROW statement" on page 884 for a description of THROW.

If you choose to handle database errors in a node, you can use the database state function SQLSTATE to receive information about the status of the DBMS call made in ESQL. You can include it in conditional statements in current node's ESQL to recognize and handle possible errors.

SQL states

In ESQL, SQL states are variable length character strings. By convention, they are six characters long and contain only the characters 0-9, A-Z . The significance of the six characters is:

Char 1

The origin of the exception

Chars 2 - 3

The class of the exception

Chars 4 - 6

The subclass of the exception

The SQL state of an exception is determined by a two stage process. In the **first stage**, the exception information is examined and any wrapping exceptions (that is, information saying what the broker was doing at the time the exception occurred) is stepped over until the exception describing the original error is located.

The **second stage** is as follows:

1. If the selected exception is a database exception, the SQL state is that supplied by the database, but prefixed by the letter "D" to avoid any confusion with exceptions arising in the broker. The SQL code, native error, and error text are those supplied by the database.
2. If the selected exception is a user exception (that is, it originated in a THROW statement), the SQL code, state, native error, and error text are taken from the first four inserts of the exception, in order. The resulting state value is taken as is (not prefixed by a letter such as "U"). In fact, the letter "U" is not used by the broker as an origin indicator. It is therefore recommended that, if you want to define a unique SQL state rather than to imitate an existing one, you use SQL states starting with the letter "U". If this recommendation is followed, it allows a handler to match all user-defined and thrown exceptions with a LIKE'U%' operator.
3. If the selected exception originated in the message transport or in the ESQL implementation itself, the SQL code, state, native error, and error text are as described in the list below.

4. For all other exceptions, the SQL state is "", indicating no origin, no class, and no subclass.

Some exceptions that currently give an empty SQL state might give individual states in future releases. If you want to catch unclassified exceptions, you are recommended to use the "all" wildcard ("%") for the SQL state on the last handler of a scope. This will continue to catch the same set of exceptions if previously unclassified exceptions are given new unique SQL states.

The following SQL states are defined:

Dddddd

dddddd is the state returned by the database.

SqlState = 'S22003'

Arithmetic overflow. An operation whose result is a numeric type resulted in a value beyond the range supported.

SqlState = 'S22004'

Null value not allowed. A null value was present in a place where null values are not allowed.

SqlState = 'S22007'

Invalid date time format. A character string used in a cast from character to a date-time type had either the wrong basic format (for example, '01947-10-24') or had values outside the ranges allowed by the Gregorian calendar (for example, '1947-21-24').

SqlState = 'S22008'

Date time field overflow. An operation whose result is a date/time type resulted in a value beyond the range supported.

SqlState = 'S22011'

SUBSTRING error. The FROM and FOR parameters, in conjunction with the length of the first operand, violate the rules of the SUBSTRING function.

SqlState = 'S22012'

Divide by zero. A divide operation whose result data type has no concept of infinity had a zero right operand.

SqlState = 'S22015'

Interval field overflow. An operation whose result is of type INTERVAL resulted in a value beyond the range supported by the INTERVAL data type.

SqlState = 'S22018'

Invalid character value for cast.

SqlState = 'SPS001'

Invalid target terminal. A PROPAGATE to terminal statement attempted to use an invalid terminal name.

SqlState = 'SPS002'

Invalid target label. A PROPAGATE to label statement attempted to use an invalid label.

SqlState = 'MQW001', SqlNativeError = 0

The bit-stream does not meet the requirements for MQ messages. No attempt was made to put it to a queue. Retrying and queue administration will not succeed in resolving this problem.

SqlState = 'MQW002', SqlNativeError = 0

The target queue or queue manager names were not valid (that is, they could

not be converted from unicode to the queue manager's code page). Retrying and queue emptying will not succeed in resolving this problem.

SqlState = 'MQW003', SqlNativeError = 0

Request mode was specified but the "reply to" queue or queue manager names were not valid (i.e. could not be converted from unicode to the message's code page). Retrying and queue emptying will not succeed in resolving this problem.

SqlState = 'MQW004', SqlNativeError = 0

Reply mode was specified but the queue or queue manager names taken from the message were not valid (that is, they could not be converted from the given code page to unicode). Retrying and queue emptying will not succeed in resolving this problem.

SqlState = 'MQW005', SqlNativeError = 0

Destination list mode was specified but the destination list supplied does not meet the basic requirements for destination lists. No attempt was made to put any message to a queue. Retrying and queue administration will not succeed in resolving this problem.

SqlState = 'MQW101', SqlNativeError = As returned by MQ

The target queue manager or queue could not be opened. Queue administration may succeed in resolving this problem but retrying will not.

SqlState = 'MQW102', SqlNativeError = as returned by MQ

The target queue manager or queue could not be written to. Retrying and queue administration might succeed in resolving this problem.

SqlState = 'MQW201', SqlNativeError = number of destinations with an error

More than one error occurred while processing a destination list. The message may have been put to zero or more queues. Retrying and queue administration might succeed in resolving this problem.

Anything that the user has used in a THROW statement

Note the recommendation to use Uuuuuu for user exceptions, unless imitating one of the exceptions defined above.

Empty string

All other errors.

ESQL datetime functions

This topic lists the ESQL datetime functions.

In addition to the functions described here, you can use arithmetic operators to perform various calculations on datetime values. For example, you can use the - (minus) operator to calculate the difference between two dates as an interval, or you can add an interval to a timestamp.

This section covers the following topics:

"EXTRACT function" on page 898

"CURRENT_DATE function" on page 900

"CURRENT_TIME function" on page 900

"CURRENT_TIMESTAMP function" on page 900

"CURRENT_GMTDATE function" on page 901

"CURRENT_GMTTIME function" on page 901

“CURRENT_GMTTIMESTAMP function” on page 901

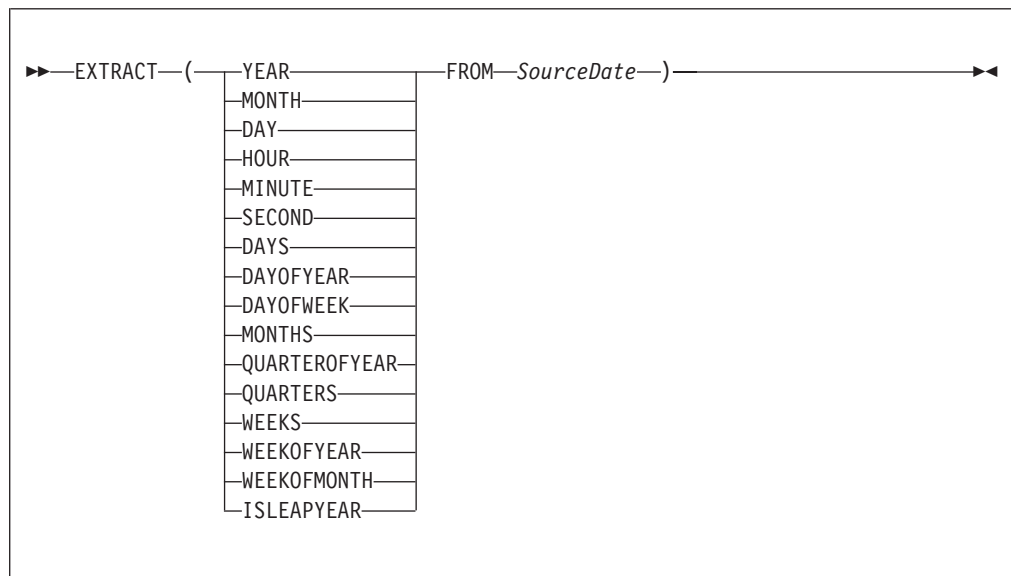
“LOCAL_TIMEZONE function” on page 902

EXTRACT function

The EXTRACT function extracts fields (or calculates values) from datetime values and intervals.

The result is INTEGER for YEAR, MONTH, DAY, HOUR, MINUTE, DAYS, DAYOFYEAR, DAYOFWEEK, MONTHS, QUARTEROFYEAR, QUARTERS, WEEKS, WEEKOFYEAR, and WEEKOFMONTH extracts, but FLOAT for SECOND extracts, and BOOLEAN for ISLEAPYEAR extracts. If the *SourceDate* is NULL, the result is NULL regardless of the type of extract.

Syntax



EXTRACT extracts individual fields from datetime values and intervals. You can extract a field only if it is present in the datetime value specified in the second parameter. Either a parse-time or a runtime error is generated if the requested field does not exist within the data type.

The following table describes the extracts that are supported in Version 6.0:

Note: All new integer values start from 1.

Table 12.

Extract	Description
YEAR	Year
MONTH	Month
DAY	Day
HOUR	Hour
MINUTE	Minute
SECOND	Second

Table 12. (continued)

Extract	Description
DAYS	Days encountered between 1st January 0001 and the <i>SourceDate</i> .
DAYOFYEAR	Day of year
DAYOFWEEK	Day of the week: Sunday = 1, Monday = 2, Tuesday = 3, Wednesday = 4, Thursday = 5, Friday = 6, Saturday = 7.
MONTHS	Months encountered between 1st January 0001 and the <i>SourceDate</i> .
QUARTEROFYEAR	Quarter of year: January to March = 1, April to June = 2, July to September = 3, October to December = 4.
QUARTERS	Quarters encountered between 1st January 0001 and the <i>SourceDate</i> .
WEEKS	Weeks encountered between 1st January 0001 and the <i>SourceDate</i> .
WEEKOFYEAR	Week of year
WEEKOFMONTH	Week of month
ISLEAPYEAR	Whether this is a leap year

Notes:

1. A week is defined as Sunday to Saturday, not any seven consecutive days. You must convert to an alternative representation scheme if required.
2. The source date time epoch is 1 January 0001. Dates before the epoch are not valid for this function.
3. The Gregorian calendar is assumed for calculation.

Example

`EXTRACT(YEAR FROM CURRENT_DATE)`

and

`EXTRACT(HOUR FROM LOCAL_TIMEZONE)`

both work without error, but

`EXTRACT(DAY FROM CURRENT_TIME)`

fails.

`EXTRACT (DAYS FROM DATE '2000-02-29')`

calculates the number of days encountered since year 1 to '2000-02-29' and

`EXTRACT (DAYOFYEAR FROM CURRENT_DATE)`

calculates the number of days encountered since the beginning of the current year but

`EXTRACT (DAYOFYEAR FROM CURRENT_TIME)`

fails because `CURRENT_TIME` does not contain date information.

CURRENT_DATE function

The CURRENT_DATE datetime function returns the current date.

Syntax

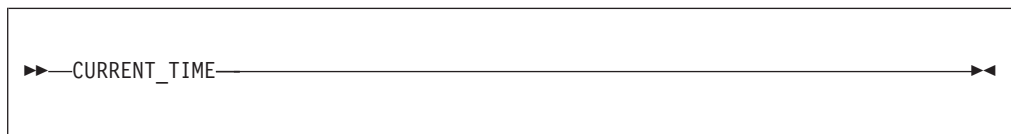


CURRENT_DATE returns a DATE value representing the current date in local time. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to CURRENT_DATE within the processing of one node are guaranteed to return the same value.

CURRENT_TIME function

The CURRENT_TIME datetime function returns the current local time.

Syntax

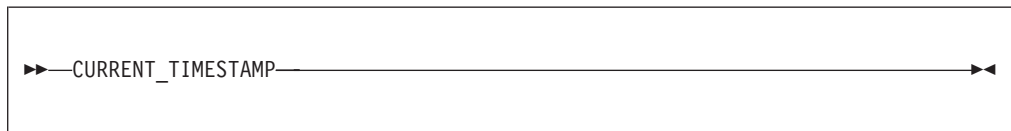


CURRENT_TIME returns a TIME value representing the current local time. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to CURRENT_TIME within the processing of one node are guaranteed to return the same value.

CURRENT_TIMESTAMP function

The CURRENT_TIMESTAMP datetime function returns the current date and local time.

Syntax



CURRENT_TIMESTAMP returns a TIMESTAMP value representing the current date and local time. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to CURRENT_TIMESTAMP within the processing of one node are guaranteed to return the same value.

Example

To obtain the following XML output message:

```
<Body>
<Message>Hello World</Message>
<DateStamp>2006-02-01 13:13:56.444730</DateStamp>
</Body>
```


use the following ESQL:

```
SET OutputRoot.XML.Body.Message = 'Hello World';
SET OutputRoot.XML.Body.DateStamp = CURRENT_TIMESTAMP;
```

CURRENT_GMTDATE function

The CURRENT_GMTDATE datetime function returns the current date in the GMT time zone.

Syntax



```
»»CURRENT_GMTDATE»»
```

CURRENT_GMTDATE returns a DATE value representing the current date in the GMT time zone. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to CURRENT_GMTDATE within the processing of one node are guaranteed to return the same value.

CURRENT_GMTTIME function

The CURRENT_GMTTIME datetime function returns the current time in the GMT time zone.

Syntax



```
»»CURRENT_GMTTIME»»
```

It returns a GMTTIME value representing the current time in the GMT time zone. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to CURRENT_GMTTIME within the processing of one node are guaranteed to return the same value.

CURRENT_GMTTIMESTAMP function

The CURRENT_GMTTIMESTAMP datetime function returns the current date and time in the GMT time zone.

Syntax

A rectangular box containing the text 'CURRENT_GMTTIMESTAMP' with a double-headed arrow pointing to the left and another double-headed arrow pointing to the right, indicating the function name and its parameters.

`CURRENT_GMTTIMESTAMP` returns a `GMTTIMESTAMP` value representing the current date and time in the GMT time zone. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to `CURRENT_GMTTIMESTAMP` within the processing of one node are guaranteed to return the same value.

LOCAL_TIMEZONE function

The `LOCAL_TIMEZONE` datetime function returns the displacement of the local time zone from GMT.

Syntax

A rectangular box containing the text 'LOCAL_TIMEZONE' with a double-headed arrow pointing to the left and another double-headed arrow pointing to the right, indicating the function name and its parameters.

`LOCAL_TIMEZONE` returns an interval value representing the local time zone displacement from GMT. As with all SQL functions that take no parameters, no parentheses are required or accepted. The value returned is an interval in hours and minutes representing the displacement of the current time zone from Greenwich Mean Time. The sign of the interval is such that a local time can be converted to a time in GMT by subtracting the result of the `LOCAL_TIMEZONE` function.

ESQL numeric functions

This topic lists the ESQL numeric functions and covers the following:

"ABS and ABSVAL functions" on page 903

"ACOS function" on page 904

"ASIN function" on page 904

"ATAN function" on page 904

"ATAN2 function" on page 904

"BITAND function" on page 905

"BITNOT function" on page 905

"BITOR function" on page 906

"BITXOR function" on page 906

"CEIL and CEILING functions" on page 907

"COS function" on page 907

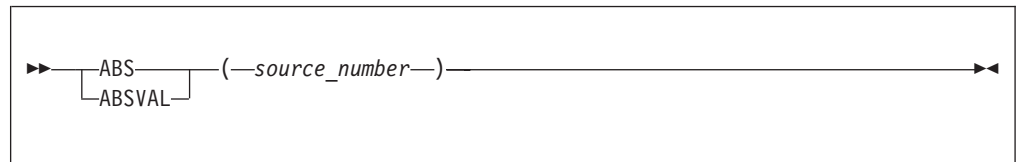
"COSH function" on page 908

- "COT function" on page 908
- "DEGREES function" on page 908
- "EXP function" on page 908
- "FLOOR function" on page 909
- "LN and LOG functions" on page 909
- "LOG10 function" on page 910
- "MOD function" on page 910
- "POWER function" on page 911
- "RADIANS function" on page 911
- "RAND function" on page 911
- "ROUND function" on page 912
- "SIGN function" on page 912
- "SIN function" on page 913
- "SINH function" on page 913
- "SQRT function" on page 913
- "TAN function" on page 914
- "TANH function" on page 914
- "TRUNCATE function" on page 915

ABS and ABSVAL functions

The ABS and ABSVAL numeric functions return the absolute value of a supplied number.

Syntax



The absolute value of the source number is a number with the same magnitude as the source but without a sign. The parameter must be a numeric value. The result is of the same type as the parameter unless it is NULL, in which case the result is NULL.

For example:

```
ABS( -3.7 )
```

```
returns 3.7
```

```
ABS( 3.7 )
```

```
returns 3.7
```

```
ABS( 1024 )
```

```
returns 1024
```

ACOS function

The ACOS numeric function returns the angle of a given cosine.

Syntax

```
▶▶ ACOS (NumericExpression) ▶▶
```

The ACOS function returns the angle, in radians, whose cosine is the given *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

ASIN function

The ASIN numeric function returns the angle of the given sine.

Syntax

```
▶▶ ASIN (NumericExpression) ▶▶
```

The ASIN function returns the angle, in radians, whose sine is the given *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

ATAN function

The ATAN numeric function returns the angle of the given tangent.

Syntax

```
▶▶ ATAN (NumericExpression) ▶▶
```

The ATAN function returns the angle, in radians, whose tangent is the given *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

ATAN2 function

The ATAN2 numeric function returns the angle subtended in a right angled triangle between an opposite and the base.

Syntax

```
▶▶ ATAN2 ( OppositeNumericExpression , BaseNumericExpression ) ▶▶
```

The ATAN2 function returns the angle, in radians, subtended (in a right angled triangle) by an opposite given by *OppositeNumericExpression* and the base given by *BaseNumericExpression*. The parameters can be any built-in numeric data type. The result is FLOAT unless either parameter is NULL, in which case the result is NULL.

BITAND function

The BITAND numeric function performs a bitwise AND on the binary representation of two or more numbers.

Syntax

```
▶▶ BITAND ( source_integer , source_integer ) ▶▶
```

BITAND takes two or more integer values and returns the result of performing the bitwise AND on the binary representation of the numbers. The result is INTEGER unless either parameter is NULL, in which case the result is NULL.

For example:

```
BITAND(12, 7)
```

returns 4 as shown by this worked example:

	Binary	Decimal
	1100	12
AND	0111	7

	0100	4

BITNOT function

The BITNOT numeric function performs a bitwise complement on the binary representation of a number.

Syntax

```
▶▶ BITNOT ( source_integer ) ▶▶
```

BITNOT takes an integer value and returns the result of performing the bitwise complement on the binary representation of the number. The result is INTEGER unless either parameter is NULL, in which case the result is NULL.

For example:

BITNOT(7)

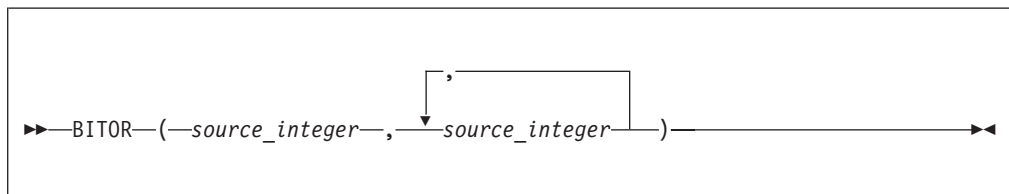
returns -8, as shown by this worked example:

	Binary	Decimal
	00...0111	7
NOT		
	<u>11...1000</u>	-8

BITOR function

The BITOR numeric function performs a bitwise OR on the binary representation of two or more numbers.

Syntax



BITOR takes two or more integer values and returns the result of performing the bitwise OR on the binary representation of the numbers. The result is INTEGER unless either parameter is NULL, in which case the result is NULL.

For example:

BITOR(12, 7)

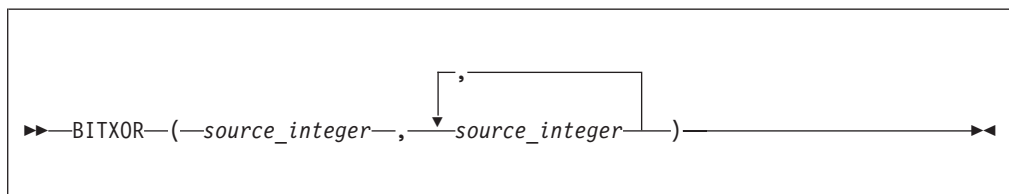
returns 15, as shown by this worked example:

	Binary	Decimal
	1100	12
OR	0111	7
	<u>1111</u>	15

BITXOR function

The BITXOR numeric function performs a bitwise XOR on the binary representation of two or more numbers.

Syntax



BITXOR takes two or more integer values and returns the result of performing the bitwise XOR on the binary representation of the numbers. The result is INTEGER unless either parameter is NULL, in which case the result is NULL.

For example:

```
BITXOR(12, 7)
```

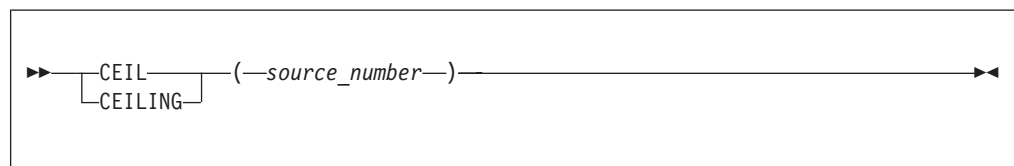
returns 11, as shown by this worked example:

Binary	Decimal
1100	12
XOR 0111	7
<hr/>	
1011	11

CEIL and CEILING functions

The CEIL and CEILING numeric functions return the smallest integer equivalent of a decimal number.

Syntax



CEIL and CEILING return the smallest integer value greater than or equal to *source_number*. The parameter can be any numeric data type. The result is of the same type as the parameter unless it is NULL, in which case the result is NULL.

For example:

```
CEIL(1)
```

returns 1

```
CEIL(1.2)
```

returns 2.0

```
CEIL(-1.2)
```

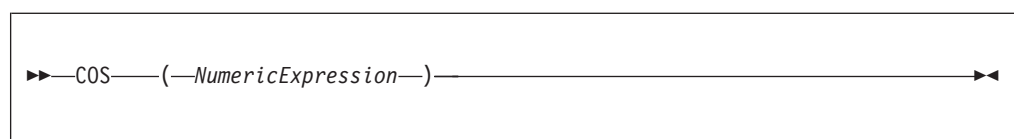
returns -1.0

If possible, the scale is changed to zero. If the result cannot be represented at that scale, it is made sufficiently large to represent the number.

COS function

The COS numeric function returns the cosine of a given angle.

Syntax



The COS function returns the cosine of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

COSH function

The COSH numeric function returns the hyperbolic cosine of a given angle.

Syntax

►►—COSH—(—*NumericExpression*—)———►◄

The COSH function returns the hyperbolic cosine of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

COT function

The COT numeric function returns the cotangent of a given angle.

Syntax

►►—COT—(—*NumericExpression*—)———►◄

The COT function returns the cotangent of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

DEGREES function

The DEGREES numeric function returns the angle of the radians supplied.

Syntax

►►—DEGREES—(—*NumericExpression*—)———►◄

The DEGREES function returns the angle, in degrees, specified by *NumericExpression* in radians. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

EXP function

The EXP numeric function returns the exponential value of a given number.

Syntax

```
►► EXP (NumericExpression) ◄◄
```

The EXP function returns the exponential of the value specified by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

FLOOR function

The FLOOR numeric function returns the largest integer equivalent to a given decimal number.

Syntax

```
►► FLOOR (source_number) ◄◄
```

FLOOR returns the largest integer value less than or equal to *source_number*. The parameter can be any numeric data type. The result is of the same type as the parameter unless it is NULL, in which case the result is NULL.

For example:

```
FLOOR(1)
```

returns 1

```
FLOOR(1.2)
```

returns 1.0

```
FLOOR(-1.2)
```

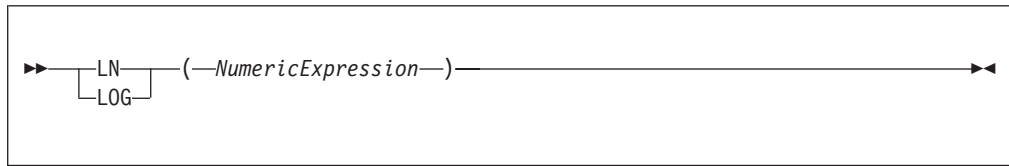
returns -2.0

If possible, the scale is changed to zero. If the result cannot be represented at that scale, it is made sufficiently large to represent the number.

LN and LOG functions

The LN and LOG equivalent numeric functions return the natural logarithm of a given value.

Syntax

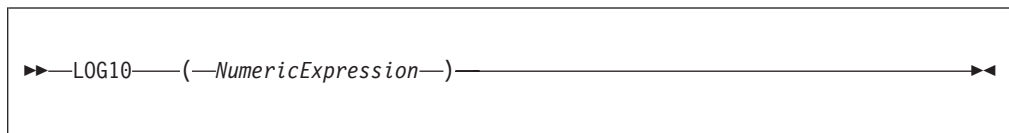


The LN and LOG functions return the natural logarithm of the value specified by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

LOG10 function

The LOG10 numeric function returns the logarithm to base 10 of a given value.

Syntax

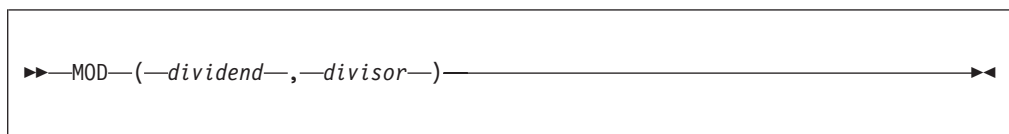


The LOG10 function returns the logarithm to base 10 of the value specified by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

MOD function

The MOD numeric function returns the remainder when dividing two numbers.

Syntax



MOD returns the remainder when the first parameter is divided by the second parameter. The result is negative only if the first parameter is negative. Parameters must be integers. The function returns an integer. If any parameter is NULL, the result is NULL.

For example:

MOD(7, 3)

returns 1

MOD(-7, 3)

returns -1

MOD(7, -3)

returns 1

MOD(6, 3)

returns 0

POWER function

The POWER numeric function raises a value to the power supplied.

Syntax

```
▶▶ POWER ( ValueNumericExpression , PowerNumericExpression ) ▶▶
```

POWER returns the given value raised to the given power. The parameters can be any built-in numeric data type. The result is FLOAT unless any parameter is NULL, in which case the result is NULL

An exception occurs, if the value is either:

- Zero and the power is negative, or
- Negative and the power is not an integer

RADIANS function

The RADIANS numeric function returns a given radians angle in degrees.

Syntax

```
▶▶ RADIANS ( NumericExpression ) ▶▶
```

The RADIANS function returns the angle, in radians, specified by *NumericExpression* in degrees. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

RAND function

The RAND numeric function returns a pseudo random number.

Syntax

```
▶▶ RAND ( IntegerExpression ) ▶▶
```

The RAND function returns a pseudo random number in the range 0.0 to 1.0. If supplied, the parameter initializes the pseudo random sequence.

The parameter can be of any numeric data type, but any fractional part is ignored. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

ROUND function

The ROUND numeric function rounds a supplied value to a given number of places.

Syntax

```
►►—ROUND—(—source_number—,—precision—)—————◄◄
```

If *precision* is a positive number, *source_number* is rounded to *precision* places right of the decimal point. If *precision* is negative, the result is *source_number* rounded to the absolute value of *precision* places to the left of the decimal point.

source_number can be any built-in numeric data type. *precision* must be an integer. The result of the function is INTEGER if the first parameter is INTEGER, FLOAT if the first parameter is FLOAT, and DECIMAL if the first parameter is DECIMAL. The result is of the same type as the *source_number* parameter unless it is NULL, in which case the result is NULL. When rounding a DECIMAL, the *banker's* or *half even symmetric* rounding rules are used. Details of these can be found in “ESQL DECIMAL data type” on page 787.

For example:

```
ROUND(27.75, 2)
```

returns 27.75

```
ROUND(27.75, 1)
```

returns 27.8

```
ROUND(27.75, 0)
```

returns 28.0

```
ROUND(27.75, -1)
```

returns 30.0

If possible, the scale is changed to the given value. If the result cannot be represented within the given scale, it is INF.

SIGN function

The SIGN numeric function tells you whether a given number is positive, negative, or zero.

Syntax

```
►►—SIGN—(—NumericExpression—)———►◄
```

The SIGN function returns -1, 0, or +1 when the *NumericExpression* value is negative, zero, or positive respectively. The parameter can be any built-in numeric data type and the result is of the same type as the parameter. If the parameter is NULL, the result is NULL.

SIN function

The SIN numeric function returns the sine of a given angle.

Syntax

```
►►—SIN—(—NumericExpression—)———►◄
```

The SIN function returns the sine of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

SINH function

The SINH numeric function returns the hyperbolic sine of a given angle.

Syntax

```
►►—SINH—(—NumericExpression—)———►◄
```

The SINH function returns the hyperbolic sine of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

SQRT function

The SQRT numeric function returns the square root of a given number.

Syntax

```
▶▶ Sqrt (—source_number—) ▶▶
```

Sqrt returns the square root of *source_number*. The parameter can be any built-in numeric data type. The result is a FLOAT. If the parameter is NULL, the result is NULL.

For example:

```
Sqrt(4)
```

returns 2E+1

```
Sqrt(2)
```

returns 1.414213562373095E+0

```
Sqrt(-1)
```

throws an exception.

TAN function

The TAN numeric function returns the tangent of a given angle.

Syntax

```
▶▶ Tan (—NumericExpression—) ▶▶
```

The TAN function returns the tangent of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

TANH function

The TANH numeric function returns the hyperbolic tangent of an angle.

Syntax

```
▶▶ Tanh (—NumericExpression—) ▶▶
```

The TANH function returns the hyperbolic tangent of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

TRUNCATE function

The TRUNCATE numeric function truncates a supplied decimal number a specified number of places.

Syntax

```
▶▶—TRUNCATE—(—source_number—,—precision—)————▶▶
```

If *precision* is positive, the result of the TRUNCATE function is *source_number* truncated to *precision* places right of the decimal point. If *precision* is negative, the result is *source_number* truncated to the absolute value of *precision* places to the left of the decimal point.

source_number can be any built-in numeric data type. *precision* must evaluate to an INTEGER. The result is of the same data type as *source_number*. If any parameter is NULL, the result is NULL.

For example:

```
TRUNCATE(27.75, 2)
```

returns 27.75

```
TRUNCATE(27.75, 1)
```

returns 27.7

```
TRUNCATE(27.75, 0)
```

returns 27.0

```
TRUNCATE(27.75, -1)
```

returns 20.0

If possible, the scale is changed to the given value. If the result cannot be represented within the given scale, it is INF.

ESQL string manipulation functions

This topic lists the ESQL string manipulation functions.

Most of the following functions manipulate all string data types (BIT, BLOB, and CHARACTER). Exceptions to this are UPPER, LOWER, LCASE, UCASE, and SPACE, which operate only on character strings.

In these descriptions, the term *singleton* refers to a single part (BIT, BLOB, or CHARACTER) within a string of that type.

In addition to the functions described here, you can use the logical OR operator to perform various calculations on ESQL string manipulation values.

To concatenate two strings, use the “ESQL string operator” on page 804.

This section covers the following topics:

“LEFT function”

“LENGTH function”

“LOWER and LCASE functions” on page 917

“LTRIM function” on page 917

“OVERLAY function” on page 918

“POSITION function” on page 919

“REPLACE function” on page 920

“REPLICATE function” on page 920

“RIGHT function” on page 921

“RTRIM function” on page 921

“SPACE function” on page 922

“SUBSTRING function” on page 922

“TRANSLATE function” on page 923

“TRIM function” on page 924

“UPPER and UCASE functions” on page 925

LEFT function

LEFT is a string manipulation function that returns a string consisting of the source string truncated to the length given by the length expression.

Syntax



```
▶▶—LEFT—(—source_string—,—LengthIntegerExpression—)————▶▶
```

The source string can be of the CHARACTER, BLOB or BIT data type and the length must be of type INTEGER. The truncation discards the final characters of the *source_string*.

The result is of the same type as the source string. If the length is negative or zero, a zero length string is returned. If either parameter is NULL, the result is NULL.

LENGTH function

The LENGTH function is used for string manipulation on all string data types (BIT, BLOB, and CHARACTER) and returns an integer value giving the number of singletons in *source_string*.

Syntax



If the *source_string* is NULL, the result is the NULL value. The term *singleton* refers to a single part (BIT, BYTE, or CHARACTER) within a string of that type.

For example:

```
LENGTH('Hello World!');
```

returns 12.


```
LENGTH('');
```

returns 0.

LOWER and LCASE functions

The LOWER and LCASE functions are equivalent, and manipulate CHARACTER string data; they both return a new character string, which is identical to *source_string*, except that all uppercase letters are replaced with the corresponding lowercase letters.

Syntax



For example:

```
LOWER('Mr Smith')
```

returns 'mr smith'.

```
LOWER('22 Railway Cuttings')
```

returns '22 railway cuttings'.

```
LCASE('ABCD')
```

returns 'abcd'.

LTRIM function

LTRIM is a string manipulation function, used for manipulating all data types (BIT, BLOB, and CHARACTER), that returns a character string value of the same data type and content as *source_string*, but with any leading default singletons removed.

Syntax

```
▶▶LTRIM(—source_string—)◀◀
```

The term *singleton* is used to refer to a single part (BIT, BLOB, or CHARACTER) within a string of that type.

The LTRIM function is equivalent to TRIM(LEADING FROM *source_string*).

If the parameter is NULL, the result is NULL.

The default singleton depends on the data type of *source_string*:

Table 13.

Character	' ' (space)
BLOB	X'00'
Bit	B'0'

OVERLAY function

OVERLAY is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER) and replaces part of a string with a substring.

Syntax

```
▶▶OVERLAY(—source_string— PLACING —source_string2—  
▶ FROM —start_position—  
└ FOR —string_length—┘ )◀◀
```

OVERLAY returns a new string of the same type as the source and is identical to *source_string*, except that a given substring in the string, starting from the specified numeric position and of the given length, has been replaced by *source_string2*. When the length of the substring is zero, nothing is replaced.

For example:

```
OVERLAY ('ABCDEFGHJIJ' PLACING '1234' FROM 4 FOR 3)
```

returns the string 'ABC1234GHIJ'

If any parameter is NULL, the result is NULL. If *string_length* is not specified, it is assumed to be equal to LENGTH(*source_string2*).

The result of the OVERLAY function is equivalent to:

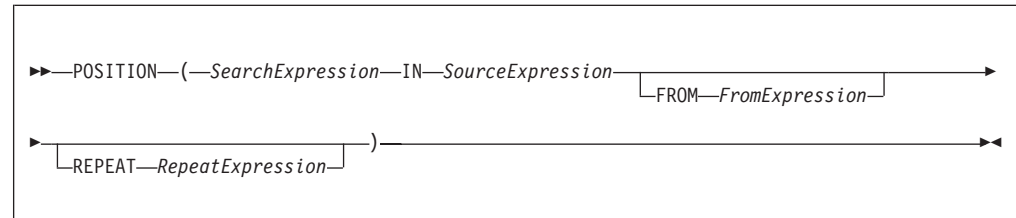
```
SUBSTRING(source_string FROM 1 FOR start_position - 1 )  
|| source_string2 ||  
SUBSTRING(source_string FROM start_position + string_length)
```

where || is the concatenation operator.

POSITION function

POSITION is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER), and returns the position of one string within another.

Syntax



POSITION returns an integer giving the position of one string (*SearchExpression*) in a second string (*SourceExpression*). A position of one corresponds to the first character of the source string.

If present, the FROM clause gives a position within the search string at which the search commences. In the absence of a FROM clause, the source string is searched from the beginning.

If present, the REPEAT clause gives a repeat count, returning the position returned to be that of the nth occurrence of the search string within the source string. If the repeat count is negative, the source string is searched from the end.

In the absence of a REPEAT clause, a repeat count of +1 is assumed; that is, the position of the first occurrence, searching from the beginning is returned. If the search string has a length of zero, the result is one.

If the search string cannot be found, the result is zero: if the FROM clause is present, this applies only to the section of the source string being searched; if the REPEAT clause is present this applies only if there are insufficient occurrences of the string.

If any parameter is NULL, the result is NULL.

The search and source strings can be of the CHARACTER, BLOB, or BIT data types but they must be of the same type.

For example:

```
POSITION('Village' IN 'Hursley Village'); returns 9
POSITION('Town' IN 'Hursley Village'); returns 0

POSITION ('B' IN 'ABCABCABCABC'); -> returns 2
POSITION ('D' IN 'ABCABCABCABC'); -> returns 0

POSITION ('A' IN 'ABCABCABCABC' FROM 4); -> returns 4
POSITION ('C' IN 'ABCABCABCABC' FROM 2); -> returns 3

POSITION ('B' IN 'ABCABCABCABC' REPEAT 2); -> returns 5
POSITION ('C' IN 'ABCABCABCABC' REPEAT 4); -> returns 12

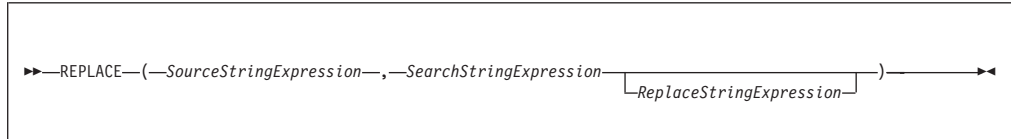
POSITION ('A' IN 'ABCABCABCABC' FROM 4 REPEAT 2); -> returns 7
POSITION ('AB' IN 'ABCABCABCABC' FROM 2 REPEAT 3); -> returns 10
```

```
POSITION ('A' IN 'ABCABCABCABC' REPEAT -2); -> returns 10
POSITION ('BC' IN 'ABCABCABCABC' FROM 2 REPEAT -3); -> returns 5
```

REPLACE function

REPLACE is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER), and replaces parts of a string with supplied substrings.

Syntax



REPLACE returns a string consisting of the source string, with each occurrence of the search string replaced by the replace string. The parameter strings can be of the CHARACTER, BLOB, or BIT data types, but all three must be of the same type.

If any parameter is NULL, the result is NULL.

The search process is single pass from the left and disregards characters that have already been matched. The following examples give the results shown:

```
REPLACE('ABCDABCDABCD', 'A', 'AA')
-- RESULT = AABCDAAABCDAAABCDAA
REPLACE('AAAABCDEFHAAAABCDEFH', 'AA', 'XYZ')
-- RESULT = XYZXYZBCDEFHXYZXYZBCDEFH
REPLACE('AAAABCDEFHAAAABCDEFH', 'AA', 'XYZ')
-- RESULT = XYZXYZABCDEFHXYZXYZBCDEFH
```

The first example shows that replacement is single pass. Each occurrence of A is replaced by AA but these are not then expanded further.

The second example shows that characters once matched are not considered further. The first AA pair is matched, replaced and disregarded. The second and third As are not matched.

The third example shows that matching is from the left. The first four As are matched as two pairs and replaced. The fifth A is not matched.

If you do not specify the replace string expression, the replace string defaults to an empty string and the behavior of the function is to delete all occurrences of the search string from the result.

REPLICATE function

REPLICATE is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER) and returns a string made up of multiple copies of a supplied string.

Syntax

```
►► REPLICATE ( PatternStringExpression , CountNumericExpression ) ◄◄
```

REPLICATE returns a string consisting of the pattern string given by *PatternStringExpression* repeated the number of times given by *CountNumericExpression*.

The pattern string can be of the CHARACTER, BLOB, or BIT datatype and the count must be of type INTEGER. The result is of the same data type as the pattern string.

If the count is negative or zero, a zero length string is returned. If either parameter is NULL, the result is NULL.

The count is limited to 32*1024*1024 to protect the broker from erroneous programs. If this limit is exceeded, an exception condition is issued.

RIGHT function

RIGHT is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER), and truncates a string.

Syntax

```
►► RIGHT ( SourceStringExpression , LengthIntegerExpression ) ◄◄
```

RIGHT returns a string consisting of the source string truncated to the length given by the length expression. The truncation discards the initial characters of the source string.

The source string can be of the CHARACTER, BLOB, or BIT data type and the length must be of type INTEGER.

If the length is negative or zero, a zero length string is returned. If either parameter is NULL, the result is NULL.

RTRIM function

RTRIM is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER), and removes trailing singletons from a string.

Syntax

```
▶▶—RTRIM—(—source_string—)————▶▶
```

RTRIM returns a string value of the same data type and content as *source_string* but with any trailing default singletons removed. The term *singleton* refers to a single part (BIT, BLOB, or CHARACTER) within a string of that type.

The RTRIM function is equivalent to TRIM(TRAILING FROM *source_string*).

If the parameter is NULL, the result is NULL.

The default singleton depends on the data type of *source_string*:

Character	' ' (space)
BLOB	X'00'
Bit	B'0'

SPACE function

SPACE is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER), and creates a string consisting of a defined number of blank spaces.

Syntax

```
▶▶—SPACE—(—NumericExpression—)————▶▶
```

SPACE returns a character string consisting of the number of blank spaces given by *NumericExpression*. The parameter must be of type INTEGER; the result is of type CHARACTER.

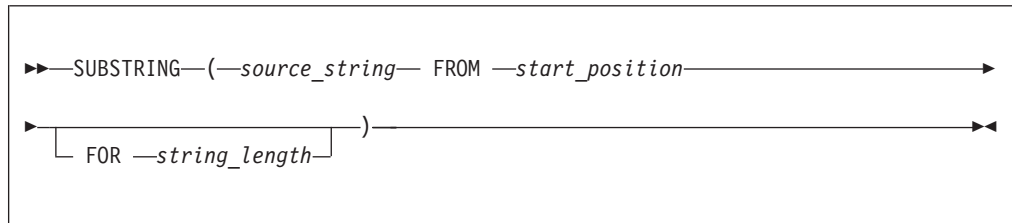
If the parameter is negative or zero, a zero length character string is returned. If the parameter is NULL, the result is NULL.

The string is limited to 32*1024*1024 to protect the broker from erroneous programs. If this limit is exceeded, an exception condition is issued.

SUBSTRING function

SUBSTRING is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER), and extracts characters from a string to create another string.

Syntax



SUBSTRING returns a new string of the same type as *source_string*, containing one contiguous run of characters extracted from *source_string* as specified by *start_position* and *string_length*.

The start position can be negative. The start position and length define a range. The result is the overlap between this range and the input string.

If any parameter is NULL, the result is NULL. This is not a zero length string.

For example:

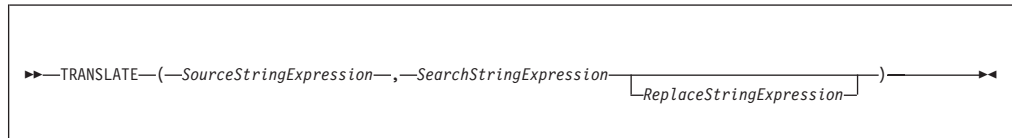
```
SUBSTRING('Hello World!' FROM 7 FOR 4)
```

returns 'Worl'.

TRANSLATE function

TRANSLATE is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER), and replaces specified characters in a string.

Syntax



TRANSLATE returns a string consisting of the source string, with each occurrence of any character that occurs in the search string being replaced by the corresponding character from the replace string.

The parameter strings can be of the CHARACTER, BLOB, or BIT data type but all three must be of the same type. If any parameter is NULL, the result is NULL.

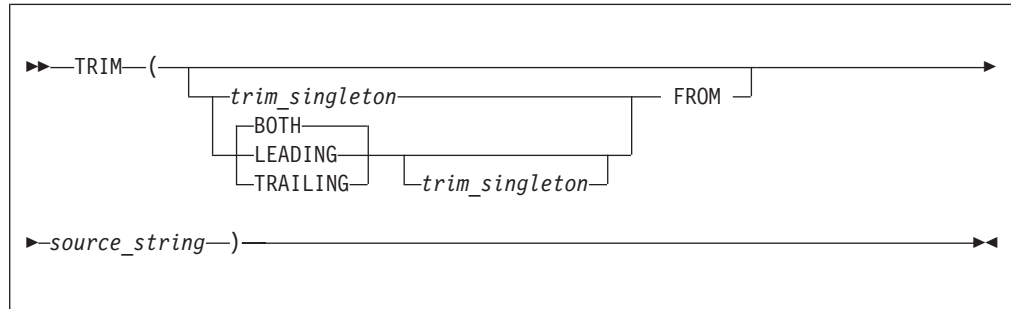
If the replace string is shorter than the search string, there are characters in the search string for which there is no corresponding character in the replace string. This is treated as an instruction to delete these characters and any occurrences of these characters in the source string are absent from the returned string.

If the replace string expression is not specified, the replace string is assumed to be an empty string, and the function deletes all occurrences of any characters in the search string from the result.

TRIM function

TRIM is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER), and removes trailing and leading singletons from a string.

Syntax



TRIM returns a new string of the same type as *source_string*, in which the leading, trailing, or both leading and trailing singletons have been removed. The term *singleton* refers to a single part (BIT, BYTE, or CHARACTER) within a string of that type.

If *trim_singleton* is not specified, a default singleton is assumed. The default singleton depends on the data type of *source_string*:

Character	' ' (space)
BLOB	X'00'
Bit	B'0'

If any parameter is NULL, the result is NULL.

It is often unnecessary to strip trailing blanks from character strings before comparison, because the rules of character string comparison mean that trailing blanks are not significant.

The following examples illustrate the behavior of the TRIM function:

```
TRIM(TRAILING 'b' FROM 'aaabB')
```

```
returns 'aaabB'.
```

```
TRIM(' a ')
```

```
returns 'a'.
```

```
TRIM(LEADING FROM ' a ')
```

```
returns 'a '.
```

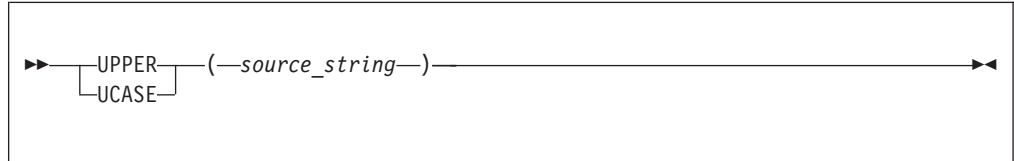
```
TRIM('b' FROM 'bbbaaabb')
```

```
returns 'aaa'.
```

UPPER and UCASE functions

UPPER and UCASE are equivalent string manipulation functions that manipulation CHARACTER string data and convert lowercase characters in a string to uppercase.

Syntax



UPPER and UCASE both return a new character string, which is identical to *source_string*, except that all lowercase letters are replaced with the corresponding uppercase letters.

For example:

```
UPPER('ABCD')
```

returns 'ABCD'.

```
UCASE('abc123')
```

returns 'ABC123'.

ESQL field functions

This topic lists the ESQL field functions and covers the following:

“ASBITSTREAM function”

“BITSTREAM function (deprecated)” on page 929

“FIELDNAME function” on page 929

“FIELDNAMESPACE function” on page 930

“FIELDTYPE function” on page 930

“FIELDVALUE function” on page 933

“FOR function” on page 933

“LASTMOVE function” on page 935

“SAMEFIELD function” on page 935

ASBITSTREAM function

The ASBITSTREAM field function generates a bit stream for the subtree of a given field according to the rules of the parser that owns the field, and uses parameters supplied by the caller for:

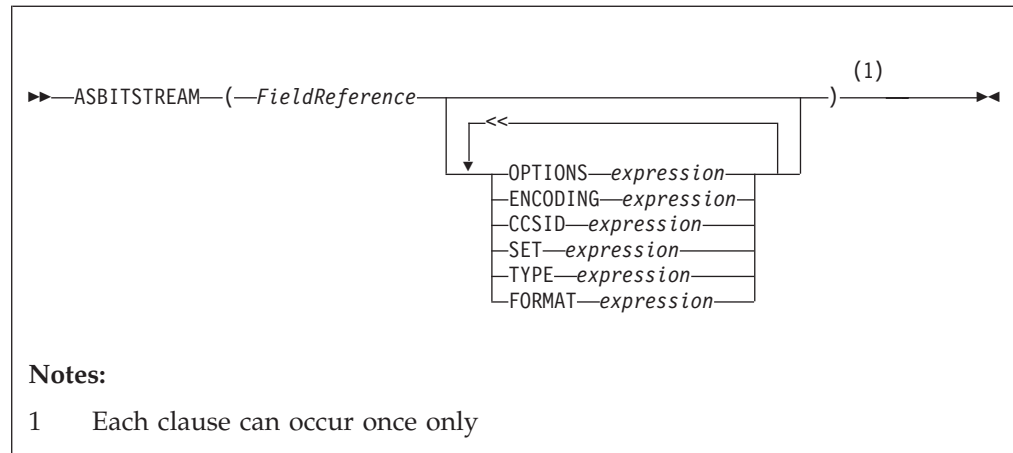
- Encoding
- CCSID
- Message set
- Message type
- Message format

- Options

This function effectively removes the limitation of the existing BITSTREAM function, which can be used only on a tree produced by a parser belonging to an input node

The BITSTREAM function is retained only for backward compatibility.

Syntax



ASBITSTREAM returns a value of type BLOB containing a bitstream representation of the field pointed to by *FieldReference* and its children

The algorithm for doing this varies from parser to parser and according to the options specified. All parsers support the following modes:

- RootBitStream, in which the bitstream generation algorithm is the same as that used by an output node. In this mode, a meaningful result is obtained only if the field pointed to is at the head of a subtree with an appropriate structure.
- EmbeddedBitStream, in which not only is the bitstream generation algorithm the same as that used by an output node, but also the
 - Encoding
 - CCSID
 - Message set
 - Message type
 - Message format

are determined, if not explicitly specified, in the same way as the output node. That is, they are determined by searching the previous siblings of *FieldReference* on the assumption that they represent headers.

In this way, the algorithm for determining these properties is essentially the same as that used for the BITSTREAM function.

Some parsers also support another mode, FolderBitStream, which generates a meaningful bit stream for any subtree, provided that the field pointed to represents a folder.

In all cases, the bit stream obtained can be given to a CREATE statement with a PARSE clause, using the same DOMAIN and OPTIONS to reproduce the original subtree.

When the function is called, any clause expressions are evaluated. An exception is thrown if any of the expressions do not result in a value of the appropriate type.

If any parameter is NULL the result is NULL.

Clause	Type	Default value
Options	integer	RootBitStream & ValidateNone
Encoding	integer	0
Ccsid	integer	0
Message set	character	Zero length string
Message type	character	Zero length string
Message format	character	Zero length string

Although the OPTIONS clause accepts any expression that returns a value of type integer, it is only meaningful to generate option values from the list of supplied constants, using the BITOR function if more than one option is required.

Once generated, the value becomes an integer and can be saved in a variable or passed as a parameter to a function, as well as being used directly in an ASBITSTREAM call. The list of globally-defined constants is:

```

Validate master options...
ValidateContentAndValue
ValidateValue    -- Can be used with ValidateContent
ValidateContent  -- Can be used with ValidateValue
ValidateNone

Validate failure action options...
ValidateException
ValidateExceptionList
ValidateLocalError
ValidateUserTrace

Validate value constraints options...
ValidateFullConstraints
ValidateBasicConstraints

Validate fix up options...
ValidateFullFixUp
ValidateNoFixUp

```

Notes:

1. The validateFullFixUp option is reserved for future use. Selecting validateFullFixUp gives identical behaviour to validateNoFixUp.
2. The validateFullConstraints option is reserved for future use. Selecting validateFullConstraints gives identical behaviour to validateBasicConstraints.
3. For full details of the validation options, refer to “Validation properties for messages in the MRM domain” on page 703.

C and Java equivalent APIs

Note that equivalent options are not available on:

- The Java plugin node API MBElement methods createElementAsLastChildFromBitstream() and toBitstream()

- The C plugin node API methods `cniCreateElementAsLastChildFromBitstream()` and `cniElementAsBitstream`.

Only one option from each group can be specified, with the exception of `ValidateValue` and `ValidateContent`, which can be used together to obtain the content and value validation. If you do not specify an option within a group, the option in bold is used.

The `ENCODING` clause accepts any expression that returns a value of type integer. However, it is only meaningful to generate encoding values from the list of supplied constants:

```
0
MQENC_INTEGER_NORMAL
MQENC_INTEGER_REVERSED
MQENC_DECIMAL_NORMAL
MQENC_DECIMAL_REVERSED
MQENC_FLOAT_IEEE_NORMAL
MQENC_FLOAT_IEEE_REVERSED
MQENC_FLOAT_S390
```

0 uses the queue manager's encoding.

The values used for the `CCSID` clause follow the normal numbering system. For example, 1200 = UCS-2, 1208 = UTF-8.

In addition the following special values are supported:

```
0
-1
```

0 uses the queue manager's `CCSID` and -1 uses the `CCSID`'s as determined by the parser itself. This value is reserved.

For absent clauses, the given default values are used. Use the `CCSID` and encoding default values, because they take their values from the queue manager's encoding and `CCSID` settings.

Similarly, use the default values for each of the message set, type, and format options, because many parsers do not require message set, type, or format information; any valid value is sufficient.

When any expressions have been evaluated, the appropriate bit stream is generated.

Note: Because this function has a large number of clauses, an alternative syntax is supported in which the parameters are supplied as a comma-separated list rather than by named clauses. In this case the expressions must be in the following order:

```
ENCODING -> CCSID -> SET -> TYPE -> FORMAT -> OPTIONS
```

The list can be truncated at any point and you can use an empty expression for any clauses for which you do not supply a value.

Examples

```
DECLARE options INTEGER BITOR(FolderBitStream, ValidateContent,  
                             ValidateValue);  
SET result = ASBITSTREAM(cursor OPTIONS options CCSID 1208);  
SET Result = ASBITSTREAM(Environment.Variables.MQRFH2.Data,,1208  
                          ,,,options);
```

BITSTREAM function (deprecated)

The BITSTREAM field function returns a value representing the bit stream described by the given field and its children. Its use is deprecated: use the newer ABITSTREAM function instead. BITSTREAM can be used only on a tree produced by a parser belonging to an input node. ABITSTREAM does not suffer from this limitation.

Syntax



The diagram shows the syntax for the BITSTREAM function. It consists of the word "BITSTREAM" followed by a hyphen, a dashed line representing a field reference, another hyphen, and a long horizontal line with arrowheads at both ends representing the rest of the function arguments.

BITSTREAM returns a value of type BLOB representing the bit stream described by the given field and its children. For incoming messages, the appropriate portion of the incoming bit stream is used. For messages constructed by Compute nodes, the following algorithm is used to establish the ENCODING, CCSID, message set, message type, and message format:

- If the addressed field has a previous sibling, and this sibling is the root of a subtree belonging to a parser capable of providing an ENCODING and CCSID, these values are obtained and used to generate the requested bit stream. Otherwise, the broker's default ENCODING and CCSID (that is, those of its queue manager) are used.
- Similarly, if the addressed field has a previous sibling, and this sibling is the root of a subtree belonging to a parser capable of providing a message set, message type, and message format, these values are obtained and used to generate the requested bit stream. Otherwise, zero length strings are used.

This function is typically used for message warehouse scenarios, where the bit stream of a message needs to be stored in a database. The function returns the bit stream of the physical portion of the incoming message, identified by the parameter. In some cases, it does not return the bit stream representing the actual field identified. For example, the following two calls return the same value:

```
BITSTREAM(Root.MQMD);  
BITSTREAM(Root.MQMD.UserIdentifier);
```

because they lie in the same portion of the message.

FIELDNAME function

The FIELDNAME field function returns the name of a given field.

Syntax

```
►►—FIELDNAME—(—source_field_reference—)—————►►
```

FIELDNAME returns the name of the field identified by *source_field_reference* as a character value. If the parameter identifies a nonexistent field, NULL is returned.

For example:

- FIELDNAME(InputRoot.XML) returns XML.
- FIELDNAME(InputBody) returns the name of the last child of InputRoot, which could be XML.
- FIELDNAME(InputRoot.*[<]) returns the name of the last child of InputRoot, which could be XML.

This function does not show any namespace information; this must be obtained by a separate call to the “FIELDNAMESPACE function.”

Whereas the following ESQL sets X to "F1":

```
SET X=FIELDNAME(InputBody.*[<]);
```

The following ESQL sets Y to null:

```
SET Y=FIELDNAME(InputBody.F1.*[<]);
```

However, the following ESQL sets Z to the (expected) child of F1:

```
SET Z=FIELDNAME(InputBody.*[<].*[<]);
```

This is because F1 belongs to a namespace and needs to be explicitly referenced by, for example:

```
DECLARE ns NAMESPACE 'urn:nid:xxxxxx';
```

```
SET Y=FIELDNAME(InputBody.ns:F1.*[<]);
```

FIELDNAMESPACE function

The FIELDNAMESPACE field function returns the namespace of a given field.

Syntax

```
►►—FIELDNAMESPACE—(—FieldReference—)—————►►
```

FIELDNAMESPACE takes a field reference as a parameter and returns a value of type CHARACTER containing the namespace of the addressed field. If the parameter identifies a nonexistent field, NULL is returned.

FIELDTYPE function

The FIELDTYPE field function returns the type of a given field.

Syntax

```
▶▶ FIELDTYPE(—source_field_reference—)▶▶
```

FIELDTYPE returns an integer representing the type of the field identified by *source_field_reference*; this is the type of the field, not the data type of the field that the parameter identifies. If the parameter identifies a nonexistent entity, NULL is returned.

The mapping of integer values to field types is not published, and might change from release to release. Compare the results of the FIELDTYPE function with named field types.

For example:

```
IF FIELDTYPE(source_field_reference) = NameValue  
THEN ...
```

The named field types that you can use in this context are listed below.

Note: The first four are domain independent; the XML.* types are applicable to the XML, XMLNS, JMSMap, and JMSStream domains, except for XML.Namespace which is specific to the XMLNS domain.

You must use these types with the capitalization shown:

- Name
- Value
- NameValue
- MQRFH2.BitStream
- XML.AsisElementContent
- XML.Attribute
- XML.AttributeDef
- XML.AttributeDefDefaultType
- XML.AttributeDefType
- XML.AttributeDefValue
- XML.AttributeList
- XML.BitStream
- XML.CDataSection
- XML.Comment
- XML.Content
- XML.DocTypeComment
- XML.DocTypeDecl
- XML.DocTypePI
- XML.DocTypeWhiteSpace
- XML.Element
- XML.ElementDef
- XML.Encoding
- XML.EntityDecl

- XML.EntityDeclValue
- XML.EntityReferenceStart
- XML.EntityReferenceEnd
- XML.ExternalEntityDecl
- XML.ExternalParameterEntityDecl
- XML.ExtSubset
- XML.IntSubset
- XML.NamespaceDecl
- XML.NotationDecl
- XML.NotationReference
- XML.ParameterEntityDecl
- XML.ParserRoot
- XML.ProcessingInstruction
- XML.PublicId
- XML.RequestedDomain
- XML.Standalone
- XML.SystemId
- XML.UnparsedEntityDecl
- XML.Version
- XML.WhiteSpace
- XML.XmlDecl
- XMLNSC.Attribute
- XMLNSC.BitStream
- XMLNSC.CDataField
- XMLNSC.CDataValue
- XMLNSC.Comment
- XMLNSC.DocumentType
- XMLNSC.DoubleAttribute
- XMLNSC.DoubleEntityDefinition
- XMLNSC.EntityDefinition
- XMLNSC.EntityReference
- XMLNSC.Field
- XMLNSC.Folder
- XMLNSC.HybridField
- XMLNSC.HybridValue
- XMLNSC.PCDataField
- XMLNSC.PCDataValue
- XMLNSC.ProcessingInstruction
- XMLNSC.SingleAttribute
- XMLNSC.SingleEntityDefinition
- XMLNSC.Value
- XMLNSC.XmlDeclaration

You can also use this function to determine whether a field in a message exists. To do this, use the form:

```
FIELDTYPE(SomeFieldReference) IS NULL
```

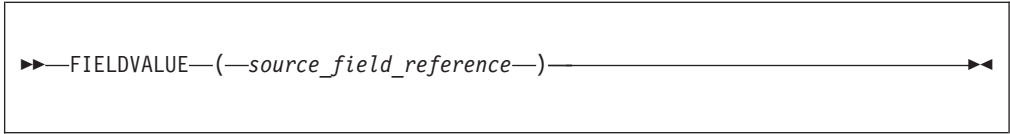
If the field exists, an integer value is returned to the function that indicates the field type (for example, string). When this is compared to NULL, the result is FALSE. If the field does not exist, NULL is returned and therefore the result is TRUE. For example:

```
IF FIELDTYPE(InputRoot.XML.Message1.Name)
  IS NULL THEN
  // Name field does not exist, take error
  action....
  ... more ESQL ...
ELSE
  // Name field does exist, continue....
  ... more ESQL ...
END IF
```

FIELDVALUE function

The FIELDVALUE field function returns the scalar value of a given field.

Syntax



```
►►—FIELDVALUE—(—source_field_reference—)—————◄◄
```

FIELDVALUE returns the scalar value of the field identified by *source_field_reference*. If it identifies a non-existent field, NULL is returned.

For example, consider the following XML input message:

```
<Data>
  <Qty Unit="Gallons">1234</Qty>
</Data>
```

The ESQL statement

```
SET OutputRoot.XML.Data.Quantity =
  FIELDVALUE(InputRoot.XML.Data.Qty);
```

gives the result:

```
<Data><Quantity>1234</Quantity></Data>
```

whereas this ESQL statement (without the FIELDVALUE function):

```
SET OutputRoot.XML.Data.Quantity =
  InputRoot.XML.Data.Qty;
```

causes a tree copy, with the result:

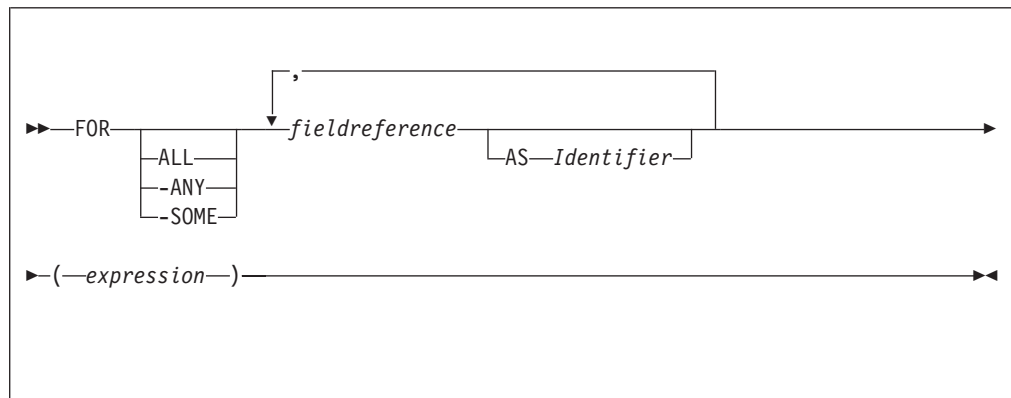
```
<Data><Quantity Unit="Gallons">1234</Quantity></Data>
```

because the field Qty is not a leaf field.

FOR function

The FOR field function evaluates an expression and assigns a resulting value of TRUE, FALSE, or UNKNOWN

Syntax



FOR enables you to write an expression that iterates over all instances of a repeating field. For each instance it processes a boolean expression and collates the results.

For example:

```
FOR ALL Body.Invoice.Purchases."Item"[] AS I (I.Quantity <= 50)
```

Note:

1. With the quantified predicate , the first thing to note is the [] on the end of the field reference after the FOR ALL. The square brackets define iteration over all instances of the Item field.

In some cases, this syntax appears unnecessary, because you can get that information from the context, but it is done for consistency with other pieces of syntax.

- 2.

The AS clause associates the name I in the field reference with the current instance of the repeating field. This is similar to the concept of iterator classes used in some object oriented languages such as C++. The expression in parentheses is a predicate that is evaluated for each instance of the Item field.

If you specify the **ALL** keyword, the function iterates over all instances of the field Item inside Body.Invoice.Purchases and evaluates the predicate I.Quantity <= 50. If the predicate evaluates to:

- TRUE (if the field is empty, or for all instances of Item) return TRUE.
- FALSE (for any instance of Item) return FALSE.
- Anything else, return UNKNOWN.

The **ANY** and **SOME** keywords are equivalent. If you use either, the function iterates over all instances of the field Item inside Body.Invoice.Purchases and evaluates the predicate I.Quantity <= 50. If the predicate evaluates to:

- FALSE (if the field is empty, or for all instances of Item) return FALSE.
- TRUE (for any instance of Item) return TRUE.
- Anything else, return UNKNOWN.

To further illustrate this, the following examples are based on the message described in "Example message" on page 991. In the following filter expression:

```
FOR ANY Body.Invoice.Purchases."Item"[] AS I (I.Title = 'The XML Companion')
```

the sub-predicate evaluates to TRUE. However, this next expression returns FALSE:

```
FOR ANY Body.Invoice.Purchases."Item"[] AS I (I.Title = 'C Primer')
```

because the C Primer is not included on this invoice. If in this instance some of the items in the invoice do not include a book title field, the sub-predicate returns UNKNOWN, and the quantified predicate returns the value UNKNOWN.

Take great care to deal with the possibility of null values appearing. Write this filter with an explicit check on the existence of the field, as follows:

```
FOR ANY Body.Invoice.Purchases."Item"[] AS I (I.Book IS NOT NULL AND  
I.Book.Title = 'C Primer')
```

The IS NOT NULL predicate ensures that, if an Item field does not contain a Book, a FALSE value is returned from the sub-predicate.

LASTMOVE function

The LASTMOVE field function tells you whether the last MOVE function succeeded.

Syntax

```
►►—LASTMOVE—(—source_dynamic_reference—)—————►◄
```

LASTMOVE returns a boolean value indicating whether the last MOVE function applied to *source_dynamic_reference* was successful (TRUE) or not (FALSE).

See “MOVE statement” on page 870 for an example of using the MOVE statement, and the LASTMOVE function to check its success.

See “Creating dynamic field references” on page 179 for information about dynamic references.

SAMEFIELD function

The SAMEFIELD field function tells you whether two field references point to the same target.

Syntax

```
►►—SAMEFIELD—(—source_field_reference1—,—source_field_reference2—)—————►◄
```

SAMEFIELD returns a BOOLEAN value indicating whether two field references point to the same target. If they do, SAMEFIELD returns TRUE; otherwise SAMEFIELD returns FALSE.

For example:

```

DECLARE ref1 REFERENCE TO OutputRoot.XML.Invoice.Purchases.Item[1];
MOVE ref1 NEXTSIBLING;
SET Result = SAMEFIELD(ref1,OutputRoot.XML.Invoice.Purchases.Item[2]);

```

Result is TRUE.

See “Creating dynamic field references” on page 179 for information about dynamic references.

ESQL list functions

This topic lists the ESQL list functions and covers the following:

“CARDINALITY function”

“EXISTS function” on page 937

“SINGULAR function” on page 937

“THE function” on page 938

CARDINALITY function

The CARDINALITY function returns the number of elements in a list.

Syntax

```

▶▶—CARDINALITY—(—ListExpression—)————▶▶

```

CARDINALITY returns an integer value giving the number of elements in the list specified by *ListExpression*.

ListExpression is any expression that returns a list. All the following, for example, return a list:

- A LIST constructor
- A field reference with the [] array indicator
- Some SELECT expressions (not all return a list)

A common use of this function is to determine the number of fields in a list before iterating over them.

Examples

```

-- Determine the number of F1 fields in the message.
-- Note that the [ ] are required
DECLARE CountF1 INT CARDINALITY(OutputRoot.XML.Data.Source.F1[]);
-- Determine the number of fields called F1 with the value 'F12' in the message.
-- Again note that the [ ] are required
DECLARE CountF1F12 INT
  CARDINALITY(SELECT F.* FROM OutputRoot.XML.Data.Source.F1[] AS F
              where F = 'F12');
-- Use the value returned by CARDINALITY to refer to a specific element
-- in a list or array:
-- Array indices start at 1, so this example refers to the third-from-last
-- instance of the Item field
Body.Invoice.Item[CARDINALITY(Body.Invoice.Item[]) - 2].Quantity

```


EXISTS function

The EXISTS function returns a BOOLEAN value indicating whether a list contains at least one element (that is, whether the list exists).

Syntax

```
►►—EXISTS—(—ListExpression—)—————►◄
```

If the list specified by *ListExpression* contains one or more elements, EXISTS returns TRUE. If the list contains no elements, EXISTS returns FALSE.

ListExpression is any expression that returns a list. All the following, for example, return a list:

- A LIST constructor
- A field reference with the [] array indicator
- Some SELECT expressions (not all return a list)

If you only want to know whether a list contains any elements or none, EXISTS executes more quickly than an expression involving the CARDINALITY function (for example, CARDINALITY(*ListExpression*) <> 0).

A common use of this function is to determine whether a field exists.

Examples

```
-- Determine whether the F1 array exists in the message. Note that the [ ]  
-- are required  
DECLARE Field1Exists BOOLEAN EXISTS(OutputRoot.XML.Data.Source.F1[]);  
-- Determine whether the F1 array contains an element with the value 'F12'.  
-- Again note that the [ ] are required  
DECLARE Field1F12Exists BOOLEAN  
  EXISTS(SELECT F.* FROM OutputRoot.XML.Data.Source.F1[] AS F where F = 'F12');
```

SINGULAR function

The SINGULAR function returns a BOOLEAN value indicating whether a list contains exactly one element.

Syntax

```
►►—SINGULAR—(—ListExpression—)—————►◄
```

If the list specified by *ListExpression* contains exactly one element, SINGULAR returns TRUE. If the list contains more or fewer elements, SINGULAR returns FALSE.

ListExpression is any expression that returns a list. All the following, for example, return a list:

- A LIST constructor
- A field reference with the [] array indicator

- Some SELECT expressions (not all return a list)

If you only want to know whether a list contains just one element or some other number, SINGULAR executes more quickly than an expression involving the CARDINALITY function (for example, `CARDINALITY(ListExpression) = 1`).

A common use of this function is to determine whether a field is unique.

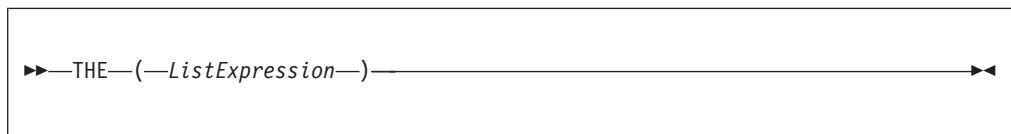
Examples

```
-- Determine whether there is just one F1 field in the message.
-- Note that the [ ] are required
DECLARE Field1Unique BOOLEAN SINGULAR(OutputRoot.XML.Data.Source.F1[]);
-- Determine whether there is just one field called F1 with the value 'F12'
-- in the message. Again note that the [ ] are required
DECLARE Field1F12Unique BOOLEAN
  SINGULAR(SELECT F.* FROM OutputRoot.XML.Data.Source.F1[] AS F where F = 'F12');
```

THE function

The THE function returns the first element of a list.

Syntax



If *ListExpression* contains one or more elements, THE returns the first element of the list. Otherwise it returns an empty list.

Restrictions

Currently, *ListExpression* must be a SELECT expression.

Complex ESQL functions

This topic lists the complex ESQL functions and covers the following:

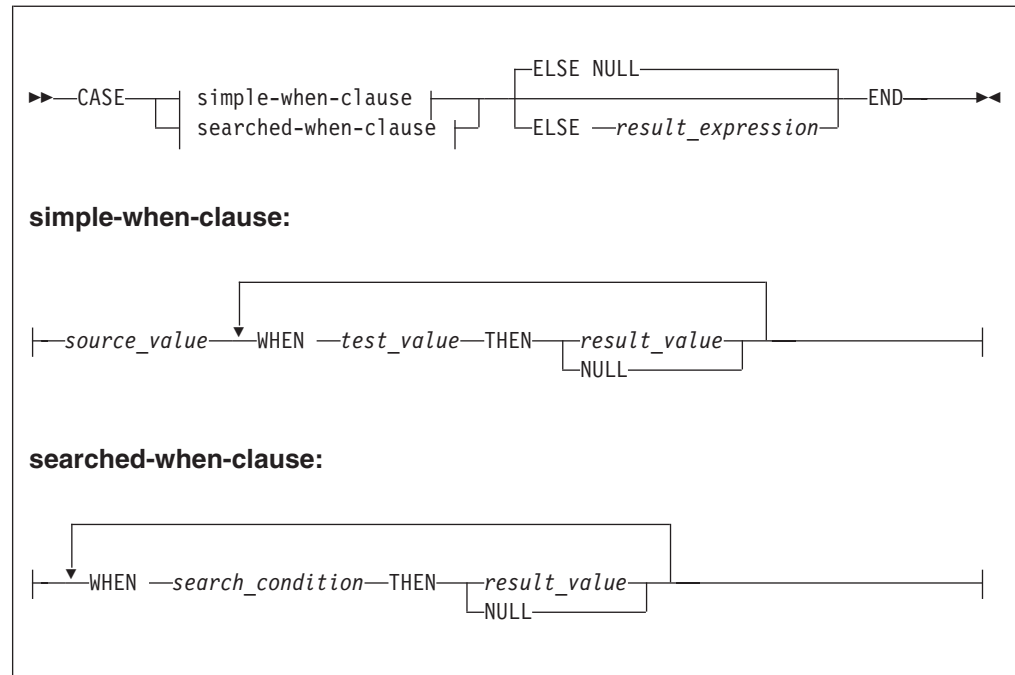
- “CASE function” on page 939
- “CAST function” on page 940
- “SELECT function” on page 954
- “ROW constructor function” on page 960
- “LIST constructor function” on page 961
- “ROW and LIST combined” on page 963
- “ROW and LIST comparisons” on page 963
- “Supported casts” on page 965
- “Implicit casts” on page 973
- “Implicit CASTs for comparisons” on page 973
- “Implicit CASTs for arithmetic operations” on page 976
- “Implicit CASTs for assignment” on page 977

“Data types of values from external sources” on page 978

CASE function

CASE is a complex function which has two forms; the simple-when form and the searched-when form. In either form CASE returns a value, the result of which controls the path of subsequent processing.

Syntax



Both forms of CASE return a value depending on a set of rules defined in WHEN clauses.

In the simple-when form, *source_value* is compared with each *test_value* until a match is found. The result of the CASE function is the value of the corresponding *result_value*. The data type of *source_value* must therefore be comparable to the data type of each *test_value*.

The CASE function must have at least one WHEN. The ELSE is optional. The default ELSE expression is NULL. A CASE expression is delimited by END. The test values do not have to be literal values.

The searched-when clause version is similar, but has the additional flexibility of allowing a number of different values to be tested.

The following example shows a CASE function with a simple WHEN clause. In this example, the CASE can be determined only by one variable that is specified next to the CASE keyword.

```
DECLARE CurrentMonth CHAR;
DECLARE MonthText CHAR;
SET CurrentMonth = SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2);

SET MonthText =
CASE CurrentMonth
WHEN '01' THEN 'January'
```

```

    WHEN '02' THEN 'February'
    WHEN '03' THEN 'March'
    WHEN '04' THEN 'April'
    WHEN '05' THEN 'May'
    WHEN '06' THEN 'June'
    ELSE 'Second half of year'
END

```

The following example shows a CASE function with a searched-when-clause. This example is still determined by one variable CurrentMonth:

```

DECLARE CurrentMonth CHAR;
DECLARE MonthText CHAR;
SET CurrentMonth = SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2);

SET MonthText =
CASE
    WHEN Month = '01' THEN 'January'
    WHEN Month = '02' THEN 'February'
    WHEN Month = '03' THEN 'March'
    WHEN Month = '04' THEN 'April'
    WHEN Month = '05' THEN 'May'
    WHEN Month = '06' THEN 'June'
    ELSE 'Second half of year'
END

```

In a searched-when-clause, different variables can be used in the WHEN clauses to determine the result. This is demonstrated in the following example of the searched-when-clause:

```

DECLARE CurrentMonth CHAR;
DECLARE CurrentYear CHAR;
DECLARE MonthText CHAR;
SET CurrentMonth = SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2);
SET CurrentYear = SUBSTRING(InputBody.Invoice.InvoiceDate FROM 1 FOR 4);

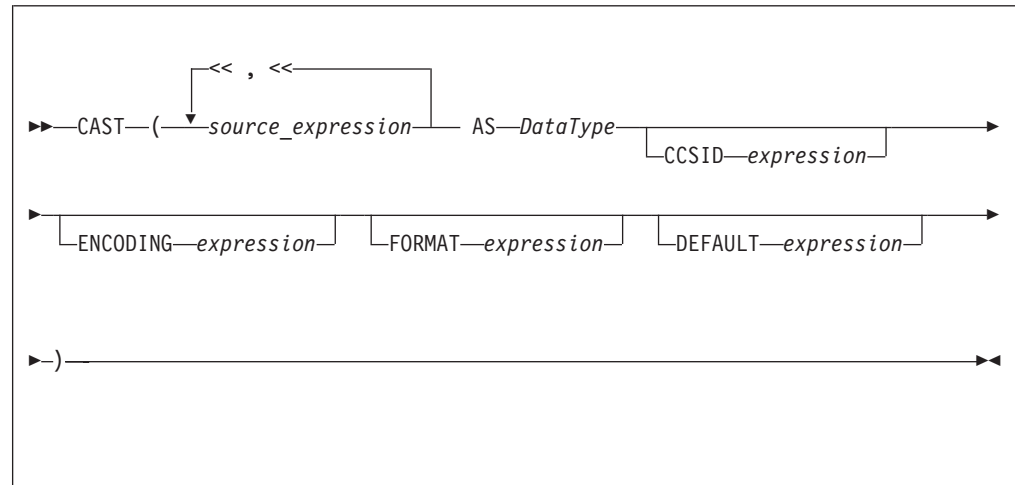
SET MonthText =
CASE
    WHEN CurrentMonth = '01' THEN 'January'
    WHEN CurrentMonth = '02' THEN 'February'
    WHEN CurrentMonth = '03' THEN 'March'
    WHEN CurrentYear = '2000' THEN 'A month in the Year 2000'
    WHEN CurrentYear = '2001' THEN 'A month in the Year 2001'
    ELSE 'Not first three months of any year or a month in the Year 2000 or 2001'
END;

```

CAST function

CAST is a complex function that transforms one or more values from one data-type into another.

Syntax



Note: In practice, you cannot specify all of the above parameters at the same time. For example, `CCSID` and `ENCODING` expressions take effect only on string-to-string conversions, while `FORMAT` applies only to string-numeric and string-datetime conversions (in either direction).

`CAST` transforms one or more values from one data-type into another data-type. For example, you can use `CAST` to process generic XML messages. All fields in an XML message have character values, so if, for example, you wanted to perform an arithmetic calculation or a date/time comparison on a field, you could use `CAST` to convert the string value of the field into a value of the appropriate type.

Not all conversions are supported; see “Supported casts” on page 965 for a list of supported conversions.

Parameters:

Source expression

`CAST` returns its first parameter (*source_expression*), which can contain more than one value, as the data-type specified by its second parameter (*DataType*). In all cases, if the source expression is `NULL`, the result is `NULL`. If the evaluated source expression is not compatible with the target data-type, or if the source expression is of the wrong format, a runtime error is generated.

CCSID

The `CCSID` clause is used only for conversions to or from one of the string data-types. It allows you to specify the code page of the source or target string.

The `CCSID` expression can be any expression evaluating to a value of type `INT`. It is interpreted according to normal WebSphere Message Broker rules for `CCSIDs`. See “Supported code pages” on page 671 for a list of valid values.

Data Type

`DataType` is the data-type into which the source value is to be transformed. The possible values are:

- String types:

- BIT
- BLOB
- CHARACTER
- Numeric types:
 - DECIMAL
 - FLOAT
 - INTEGER
- Date/Time types:
 - DATE
 - GMTTIME
 - GMTTIMESTAMP
 - INTERVAL
 - TIME
 - TIMESTAMP
- Boolean:
 - BOOLEAN

DEFAULT

The DEFAULT clause provides a method of avoiding exceptions being thrown from CAST statements by providing a last-resort value to return.

The DEFAULT *expression* must be a valid ESQL expression that returns the same data-type as that specified on the **Data Type** parameter, otherwise an exception is thrown.

The CCSID, ENCODING, and FORMAT parameters are not applied to the result of the DEFAULT expression; the expression must, therefore, be of the correct CCSID, ENCODING, and FORMAT.

ENCODING

The ENCODING clause allows you to specify the encoding. It is used for certain conversions only. The ENCODING value can be any expression evaluating to a value of type INT. It is interpreted according to normal WebSphere Message Broker rules for encoding. Valid values are:

- MQENC_NATIVE (0x00000222L)
- MQENC_INTEGER_NORMAL (0x00000001L)
- MQENC_INTEGER_REVERSED (0x00000002L)
- MQENC_DECIMAL_NORMAL (0x00000010L)
- MQENC_DECIMAL_REVERSED (0x00000020L)
- MQENC_FLOAT_IEEE_NORMAL (0x00000100L)
- MQENC_FLOAT_IEEE_REVERSED (0x00000200L)
- MQENC_FLOAT_S390 (0x00000300L)

FORMAT

For conversions between string data-types and numerical or date-time data-types, you can supply an optional FORMAT expression. For conversions *from* string types, FORMAT defines how the source string should be parsed to fill the target data-type. For conversions *to* string types, it defines how the data in the source expression is to be formatted in the target string.

FORMAT takes different types of expression for date-time and numerical conversions. However, the same FORMAT expression can be used irrespective of whether the conversion is to a string or from a string.

You can specify a FORMAT expression when casting:

- From any of the string data-types (BIT, BLOB, or CHARACTER) to:
 - DECIMAL
 - FLOAT
 - INTEGER
 - DATE
 - GMTTIMESTAMP
 - TIMESTAMP
 - GMTTIME
 - TIME
- To any of the string data-types (BIT, BLOB, or CHARACTER) from any of the numerical and date-time data-types listed in the previous bullet.

Specifying FORMAT for an unsupported combination of source and target data-types causes error message BIP3205 to be issued.

For more information about conversion to and from numerical data-types see “Formatting and parsing numbers as strings” on page 945. For more information about conversion to and from date-time data-types, see “Formatting and parsing dateTimes as strings” on page 948.

The FORMAT expression is equivalent to those used in many other products, such as ICU and Microsoft Excel.

Examples:

Example 1. Formatted CAST from DECIMAL to CHARACTER

```
DECLARE source DECIMAL 31415.92653589;
DECLARE target CHARACTER;
DECLARE pattern CHARACTER '#,##0.00';
SET target = CAST(source AS CHARACTER FORMAT pattern);
-- target is now "31,415.93"
```

Example 2. Formatted CAST from DATE to CHARACTER

```
DECLARE now CHARACTER = CAST(CURRENT_TIMESTAMP AS CHARACTER
                             FORMAT "yyyyMMdd-HHmss");
-- target is now "20041007-111656" (in this instance at least)
```

Example 3. Formatted CAST from CHARACTER to DATE

```
DECLARE source CHARACTER '01-02-03';
DECLARE target DATE;
DECLARE pattern CHARACTER 'dd-MM-yy';
SET target = CAST(source AS DATE FORMAT pattern);
-- target now contains Year=2003, Month=02, Day=01
```

Example 4. Formatted CAST from CHARACTER to TIMESTAMP

```
DECLARE source CHARACTER '12 Jan 03, 3:45pm';
DECLARE target TIMESTAMP;
DECLARE pattern CHARACTER 'dd MMM yy, h:mma';
SET target = CAST(source AS TIMESTAMP FORMAT pattern);
-- target now contains Year=2003, Month=01, Day=03, Hour=15, Minute=45,
Seconds=58
-- (seconds taken from CURRENT_TIME since not present in input)
```

Example 5. Formatted CAST from DECIMAL to CHARACTER, with negative pattern

```
DECLARE source DECIMAL -54231.122;
DECLARE target CHARACTER;
DECLARE pattern CHARACTER '#,##0.00;(#,##0.00)';
SET target = CAST(source AS CHARACTER FORMAT pattern);
-- target is now "£(54,231.12)"
```

Example 6. Formatted CAST from CHARACTER to TIME

```
DECLARE source CHARACTER '16:18:30';
DECLARE target TIME;
DECLARE pattern CHARACTER 'hh:mm:ss';
SET target = CAST(source AS TIME FORMAT pattern);
-- target now contains Hour=16, Minute=18, Seconds=30
```

Example 7. CASTs from the numeric types to DATE

```
CAST(7, 6, 5 AS DATE);
CAST(7.4e0, 6.5e0, 5.6e0 AS DATE);
CAST(7.6, 6.51, 5.4 AS DATE);
```

Example 8. CASTs from the numeric types to TIME

```
CAST(9, 8, 7 AS TIME);
CAST(9.4e0, 8.6e0, 7.1234567e0 AS TIME);
CAST(9.6, 8.4, 7.7654321 AS TIME);
```

Example 9. CASTs from the numeric types to GMTTIME

```
CAST(DATE '0001-02-03', TIME '04:05:06' AS TIMESTAMPTIME);
CAST(2, 3, 4, 5, 6, 7.8 AS TIMESTAMPTIME);
```

Example 10. CASTs to TIMESTAMPTIME

```
CAST(DATE '0001-02-03', TIME '04:05:06' AS TIMESTAMPTIME);
CAST(2, 3, 4, 5, 6, 7.8 AS TIMESTAMPTIME);
```

Example 11. CASTs to GMTTIMESTAMPTIME

```
CAST(DATE '0002-03-04', GMTTIME '05:06:07' AS GMTTIMESTAMPTIME);
CAST(3, 4, 5, 6, 7, 8 AS GMTTIMESTAMPTIME);
CAST(3.1e0, 4.2e0, 5.3e0, 6.4e0, 7.5e0, 8.6789012e0 AS GMTTIMESTAMPTIME);
CAST(3.2, 4.3, 5.4, 6.5, 7.6, 8.7890135 AS GMTTIMESTAMPTIME);
```

Example 12. CASTs to INTERVAL from INTEGER

```
CAST(1234 AS INTERVAL YEAR);
CAST(32, 10 AS INTERVAL YEAR TO MONTH);
CAST(33, 11 AS INTERVAL DAY TO HOUR);
CAST(34, 12 AS INTERVAL HOUR TO MINUTE);
CAST(35, 13 AS INTERVAL MINUTE TO SECOND);
CAST(36, 14, 10 AS INTERVAL DAY TO MINUTE);
CAST(37, 15, 11 AS INTERVAL HOUR TO SECOND);
CAST(38, 16, 12, 10 AS INTERVAL DAY TO SECOND);
```

Example 13. CASTs to INTERVAL from FLOAT

```
CAST(2345.67e0 AS INTERVAL YEAR);
CAST(3456.78e1 AS INTERVAL MONTH);
CAST(4567.89e2 AS INTERVAL DAY);
CAST(5678.90e3 AS INTERVAL HOUR);
CAST(6789.01e4 AS INTERVAL MINUTE);
CAST(7890.12e5 AS INTERVAL SECOND);
CAST(7890.1234e0 AS INTERVAL SECOND);
```


Example 14. CASTs to INTERVAL from DECIMAL

```
CAST(2345.67 AS INTERVAL YEAR );
CAST(34567.8 AS INTERVAL MONTH );
CAST(456789 AS INTERVAL DAY );
CAST(5678900 AS INTERVAL HOUR );
CAST(67890100 AS INTERVAL MINUTE);
CAST(789012000 AS INTERVAL SECOND);
CAST(7890.1234 AS INTERVAL SECOND);
```

Example 15. CASTs to FLOAT from INTERVAL

```
CAST(INTERVAL '1234' YEAR AS FLOAT);
CAST(INTERVAL '2345' MONTH AS FLOAT);
CAST(INTERVAL '3456' DAY AS FLOAT);
CAST(INTERVAL '4567' HOUR AS FLOAT);
CAST(INTERVAL '5678' MINUTE AS FLOAT);
CAST(INTERVAL '6789.01' SECOND AS FLOAT);
```

Example 16. CASTs DECIMAL from INTERVAL

```
CAST(INTERVAL '1234' YEAR AS DECIMAL);
CAST(INTERVAL '2345' MONTH AS DECIMAL);
CAST(INTERVAL '3456' DAY AS DECIMAL);
CAST(INTERVAL '4567' HOUR AS DECIMAL);
CAST(INTERVAL '5678' MINUTE AS DECIMAL);
CAST(INTERVAL '6789.01' SECOND AS DECIMAL);
```

Example 17. A ternary cast that fails and results in the substitution of a default value

```
CAST(7, 6, 32 AS DATE DEFAULT DATE '1947-10-24');
```

Example 18. A sexternary cast that fails and results in the substitution of a default value

```
CAST(2, 3, 4, 24, 6, 7.8 AS TIMESTAMP DEFAULT TIMESTAMP '1947-10-24 07:08:09');
```

Example 19. A ternary cast that fails and throws an exception

```
BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE LIKE '%' BEGIN
    SET OutputRoot.XML.Data.Date.FromIntegersInvalidCast = 'Exception thrown';
  END;

  DECLARE Dummy CHARACTER CAST(7, 6, 32 AS DATE);
END;
```

Example 20. A sexternary cast that fails and throws an exception

```
BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE LIKE '%' BEGIN
    SET OutputRoot.XML.Data.Timestamp.FromIntegersInvalidCast = 'Exception thrown';
  END;

  DECLARE Dummy CHARACTER CAST(2, 3, 4, 24, 6, 7.8 AS TIMESTAMP);
END;
```

Formatting and parsing numbers as strings:

For conversions between string data-types and numerical data-types, you can supply, on the `FORMAT` parameter of the `CAST` function, an optional formatting expression. For conversions *from* string types, the formatting expression defines how the source string should be parsed to fill the target data-type. For conversions *to* string types, it defines how the data in the source expression is to be formatted in the target string.

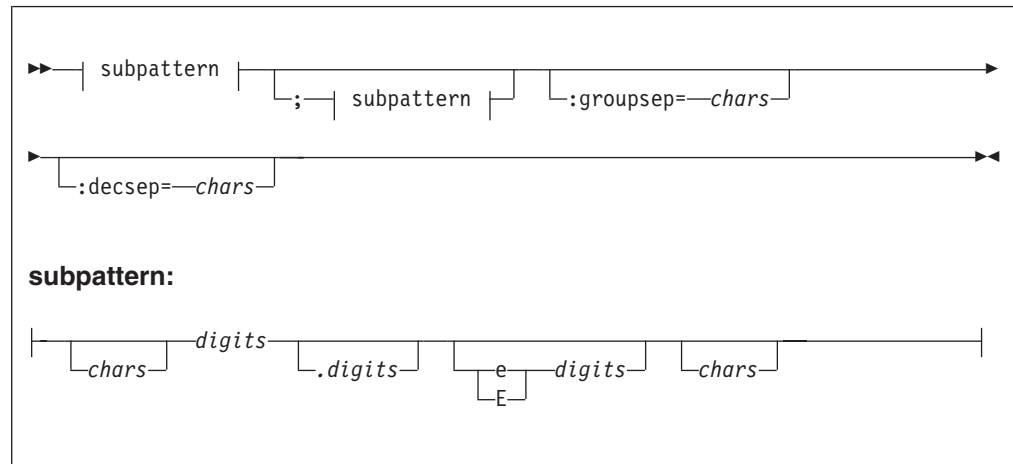
You can specify a FORMAT expression for the following numerical conversions. (Specifying a FORMAT expression for date/time conversions is described in “Formatting and parsing dateTimes as strings” on page 948.)

- From any of the string data-types (BIT, BLOB, or CHARACTER) to:
 - DECIMAL
 - FLOAT
 - INTEGER
- To any of the string data-types (BIT, BLOB, or CHARACTER) from any of the numerical data-types listed in the previous bullet.

The formatting expression consists of three parts:

1. A subpattern defining positive numbers.
2. An optional subpattern defining negative numbers. (If only one subpattern is defined, negative numbers use the positive pattern, prefixed with a minus sign.)
3. The optional parameters groupsep and decsep.

Syntax



Parameters:
chars

A sequence of zero or more characters. All characters can be used, *except* the special characters listed in Table 14 on page 947.

decsep

One or more characters to be used as the separator between the whole and decimal parts of a number (the decimal separator). The default decimal separator is a period (.).

digits

A sequence of one or more of the numeric tokens (0 # - + , .) listed in Table 14 on page 947.

groupsep

One or more characters to be used as the separator between clusters of integers, to make large numbers more readable (the grouping separator). The default grouping separator is nothing (that is, there is no grouping of digits or separation of groups).

Grouping is commonly done in thousands, but it can be redefined by either the pattern or the locale. There are two grouping sizes:

The primary grouping size

Used for the least significant integer digits.

The secondary grouping size

Used for all other integer digits.

In most cases, the primary and secondary grouping sizes are the same, but can be different. For example, if the pattern used is `#,##,##0`, the primary grouping size is 3 and the secondary is 2. The number 123456789 would become the string `"12,34,56,789"`.

If multiple grouping separators are used (as in the previous example), the rightmost separator defines the primary size and the penultimate rightmost separator defines the secondary size.

subpattern

The subpattern consists of:

1. An optional prefix (*chars*)
2. A mandatory pattern representing a whole number
3. An optional pattern representing decimal places
4. An optional pattern representing an exponent (the power by which the preceding number is raised)
5. An optional suffix (*chars*)

Parts 2, 3, and 4 of the subpattern are defined by the tokens in the following table.

Table 14. Tokens to define a formatting subpattern used for numeric/string conversions

Token	Represents
0	Any digit, including a leading zero.
#	Any digit, excluding a leading zero. (See the explanation of the difference between 0 and # that follows this table.)
.	Decimal separator.
+	Prefix of positive numbers.
-	Prefix of negative numbers.
,	Grouping separator.
E/e	Separates the number from the exponent.
;	Subpattern boundary.
'	Quote, used to quote special characters. If a quote is needed in output, it must be doubled (").
*	Padding specifier. The character following the asterisk is used to pad the number to fit the length of the format pattern.

The # and 0 characters are used for digit substitution, the difference between them being that a # character is removed if there is no number to replace it with. For example, 10 formatted by the pattern #,##0.00 gives "10.00", but formatted by 0,000.00 gives "0,010.00".

To specify padding characters, use an asterisk. When an asterisk is placed in either of the two *chars* regions (the prefix and suffix), the character immediately following it is used to pad the output. Padding can be specified only once. For example, a pattern of *x#,###,##0.00 applied to 1234 would give "xxx1,234.00". Applied to 1234567, it would give "1,234,567.00".

Examples of formatting patterns:

Table 15. Examples of formatting patterns, showing the strings output from sample numerical input

Pattern	Input number	Output string
+++ ,##0.00;- ###,###,##0.00:groupsep='':decsep=,	123456789.123	"+123'456'789,12"
##0.00	1000000	"1000000.00"
##0.00	3.14159265	"3.14"

Formatting and parsing dateTimes as strings:

This section gives information on how you can specify the dateTime format using a string of pattern letters.

When you are converting a date or time into a string, a format pattern can be applied that directs the conversion. This would apply if you were formatting from a date or time into a string or parsing a string into a date or time.

During the formatting (for example, a dateTime to a string) a pattern or a set of tokens is replaced with their equivalent source. Figure 1 gives a representation of how this is applied.

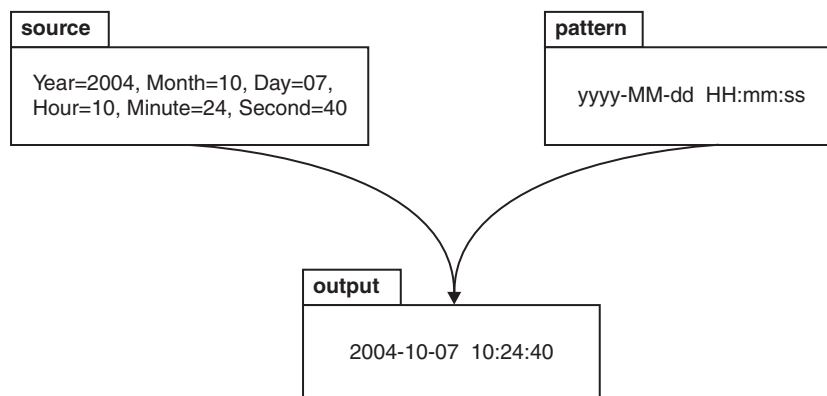


Figure 1. Using a pattern to format a dateTime source to produce a string output.

When a string is parsed (for example, converting it to a dateTime), the pattern or set of tokens are used to determine which part of the target dateTime is

represented by which part of the string. Figure 2 gives a representation of how this is applied.

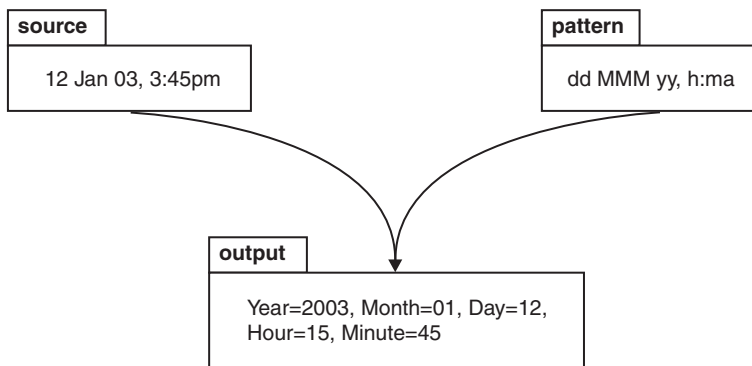
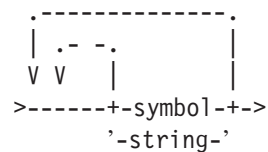


Figure 2. Using a pattern to parse a string source to produce a date`Time` output.

Syntax

The expression pattern is defined by:



Where:

symbol

is a character in the set `adDeEFGhHIkKmMsSTUwWyYzZ`.

string is a sequence of characters enclosed in single quotation marks. If a single quote is required within the string, two single quotes, `''`, can be used.

Characters for formatting a date`Time` as a string

The following table lists the allowable characters that can be used in a pattern for formatting or parsing strings in relation to a date`Time`.

Symbol	Meaning	Presentation	Examples
a	am or pm marker	Text	Input am, AM, pm, PM. Output AM or PM
d	day in month (1-31)	Number	1, 20
dd	day in month (01-31)	Number	01, 31
D	day in year (1-366)	Number	3, 80, 100
DD	day in year (01-366)	Number	03, 80, 366
DDD	day in year (001-366)	Number	003
e	day in week (1-7)	Number	2 ⁶
EEE	day in week	Text	Tue
EEEE	day in week	Text	Tuesday

Symbol	Meaning	Presentation	Examples
F	day of week in month (1-5)	Number	2 (for second Wednesday in July) ³
G	Era	Text	BC or AD
h	hour in am or pm (1-12)	Number	6
hh	hour in am or pm (01-12)	Number	06
H	hour of day in 24 hour form (0-23)	Number	7 ⁷
HH	hour of day in 24 hour form (00-23)	Number	07 ⁷
I	ISO8601 Date/Time (up to yyyy-MM-dd'T'HH:mm:ss.SSSZZZ)	text	2004-10-07T12:06:56.568+01:00 ⁵
IU	ISO8601 Date/Time (as above, but ZZZ with output "Z" if the timezone is +00:00)	text	2004-10-07T12:06:56.568+01:00, 2003-12-15T15:42:12.000Z ⁵
k	hour of day in 24 hour form (1-24)	Number	8 ⁷
k	hour of day in 24 hour form (01-24)	Number	08 ⁷
K	hour in am or pm (0-11)	Number	9
KK	hour in am or pm (00-11)	Number	09
m	minute	Number	4
mm	minute	Number	04
M	numeric month	Number	5, 12
MM	numeric month	Number	05, 12
MMM	named month	Text	Jan, Feb
MMMM	named month	Text	January, February
s	seconds	Number	5
ss	seconds	Number	05
S	decisecond	Number	7 ⁸
SS	centisecond	Number	70 ⁸
SSS	millisecond	Number	700 ⁸
SSSS	1/10,000 th seconds	Number	7000 ⁸
SSSSS	1/100,000 th seconds	Number	70000 ⁸
SSSSSS	1/1,000,000 th seconds	Number	700000 ⁸
T	ISO8601 Time (up to HH:mm:ss.SSSZZZ)	text	12:06:56.568+01:00 ⁵
TU	ISO8601 Time (as above, but a timezone of +00:00 is replaced with 'Z')	text	12:06:56.568+01:00, 15:42:12.000Z ⁵
w	week in year	Number	7, 53 ²
ww	week in year	Number	07, 53 ²
W	week in month	Number	2 ⁴
yy	year	Number	96 ¹

Symbol	Meaning	Presentation	Examples
yyyy	year	Number	1996 ¹
YY	year: use with week in year only	Number	96 ²
YYYY	year: use with week in year only	Number	1996 ²
zzz	time zone (abbreviated name)	Text	gmt
zzzz	time zone (full name)	Text	Greenwich Mean Time
Z	time zone (+/-n)	Text	+3
ZZ	time zone (+/-nn)	Text	+03
ZZZ	time zone (+/-nn:nn)	Text	+03:00
ZZZU	time zone (as ZZZ, "+00:00" is replaced by "Z")	Text	+03:00, Z
ZZZZ	time zone (GMT+/-nn:nn)	Text	GMT+03:00
ZZZZZ	time zone (as ZZZ, but no colon) (+/-nnnn)	Text	+0300
'	escape for text		'User text'
"	(two single quotes) single quote within escaped text		'o'clock'

The presentation of the date`Time` object depends on what symbols you specify as follows:

- **Text.** If you specify four or more of the symbols, the full form is presented. If you specify less than four, the short or abbreviated form, if it exists, is presented. For example, EEEE produces Monday, EEE produces Mon.
- **Number.** The number of characters for a numeric date`Time` component must be within the bounds of the corresponding formatting symbols. Repeat the symbol to specify the minimum number of digits required. The maximum number of digits permitted will be the upper bound for a particular symbol. For example, day in month has an upper bound of 31 therefore a format string of d will allow the values of 2 or 21 to be parsed but will disallow the value of 32 or 210. On output, numbers are padded with zeros to the specified length. A year is a special case, see note 1 in the list below. Fractional seconds are also special case, see note 8 below.
- Any characters in the pattern that are not in the ranges of [`'a'..'z'`] and [`'A'..'Z'`] are treated as quoted text. For example, characters like colon (:), comma (,), period (.), the number sign (hash or pound, #), the at sign (@) and space appear in the resulting time text even if they are not enclosed within single quotes.
- You can create formatting strings that produce unpredictable results, so you must use these symbols with care. For example, if you specify dMyyyy, it is impossible to distinguish between day, month, and year. dMyyyy tells the broker that a minimum of one character represents the day, a minimum of one character represents the month, and four characters represent the year. Therefore 3111999 could be interpreted as 3/11/1999 and 31/1/1999.

Notes: The following points explain the notes in the table above:

1. Year is handled as a special case:

- On output, if the count of *y* is 2, the year is truncated to 2 digits. For example, if *yyyy* produces 1997, *yy* produces 97.
 - On input, for 2 digits years the century window is fixed to 53. For example an input date of 52 will result in a year value of 2052, whilst 53 would give an output year of 1953 and 97 would give 1997.
2. In ESQL, the first day of the year is assumed to be in the first week, thus January 1 is always in week 1. This can lead to dates specified relative to one year actually being in a different year. For example, "Monday week 1 2005" parsed using "EEEE' week 'w' 'YYYY" would give a date of 2004-12-27, since the Monday of the first week in 2005 is actually a date in 2004.
If you use the *y* symbol, the adjustment is not done and unpredictable results could occur for dates around the end of the year. For example, if the string "2005 01 Monday" is formatted:
 - Monday of week 1 in 2005 using format string "YYYY ww EEEE" is correctly interpreted as 27st December 2004
 - Monday of week 1 in 2005 using format string "yyyy ww EEEE" is incorrectly interpreted as 27th December 2005
 3. The 11th July 2001 is the second Wednesday in July and can be expressed as 2001 July Wednesday 2 using format string *yyyy MMM EEEE F*. This is not the same as Wednesday in week 2 of July 2001, which is 4th July 2001.
 4. The first and last week in a month might include days from neighboring months. For example, Tuesday 31st July 2001 could be expressed as *Tuesday in week one of August 2001*, which is 2001 08 1 Tuesday using format string *yyyy MM W EEEE*.
 5. See the section *ISO8601, I and T DateTime tokens*.
 6. The values specified in the day in week field are fixed to:
 - 1 - Sunday
 - 2 - Monday
 - 3 - Tuesday
 - 4 - Wednesday
 - 5 - Thursday
 - 6 - Friday
 - 7 - Saturday
 7. 24 hour fields may result in an ambiguous time if specified with a conflicting am/pm field.
 8. Fractional Seconds The length must implicitly match the number of format symbols on input. The output is rounded to the specified length.
 9. Long time zones work best when used in the Continent/City format. Similarly, on Unix systems, the TZ environment variable should be specified using the Continent/City format.

ISO8601, I and T DateTime tokens

If your *dateTime* values are compliant with the ISO8601:2000 'Representation of dates and times' standard, you should consider whether it is possible to use the formatting symbols I and T. These match a subset of the ISO8601 standard, specifically:

- The restricted profile as proposed by the W3C at <http://www.w3.org/TR/NOTE-datetime>

- Truncated representations of calendar dates as specified in section 5.2.1.3 of ISO8601:2000
 - Basic format (sub-sections c, e and f)
 - Extended format (sub-sections a, b and d)

These symbols should only be used on their own.

- The I formatting symbol matches any dateTime string conforming to the supported subset.
- The T formatting symbol matches any dateTime string conforming to the supported subset that consists of a time portion only.

On output, following form will be applied depending on the logical datatype:

Logical MRM Datatype	Logical ESQL Datatype	Output Form
xsd:dateTime	TIMESTAMP or GMTTIMESTAMP	yyyy-MM-dd'T'HH:mm:ss.SSSZZZ
xsd:date	DATE	yyyy-MM-dd
xsd:gYear	INTERVAL	yyyy
xsd:gYearMonth	INTERVAL	yyyy-MM
xsd:gMonth	INTERVAL	--MM
xsd:gmonthDay	INTERVAL	--MM-dd
xsd:gDay	INTERVAL	---dd
xsd:time	TIME / GMTTIME	'T'HH:mm:ss.SSSZZZ

Note:

- On input both I and T accept '+00:00' and 'Z' to indicated a zero time difference from Coordinated Universal Time (UTC), but on output will always generate '+00:00'. If you require that 'Z' is always generated on output, you should use the alternative IU or TU formatting symbols instead.
- ZZZ will always output '+00:00' to indicate a zero time difference from Coordinated Universal Time (UTC). If you require that 'Z' is always generated on output, you should use the alternative ZZZU form instead.

Using the input UTC format on output

An element or attribute of logical type xsd:dateTime or xsd:time that contains a dateTime as a string can specify Consolidated Universal Time (UTC) by using either the Z character or time zone +00:00. On input the MRM parser remembers the UTC format of such elements and attributes. On output you can specify whether Z or +00:00 should appear by using the dateTime format property of the element or attribute. Alternatively you can preserve the input UTC format by checking message set property Use input UTC format on output. If this property is checked, then the UTC format will be preserved into the output message and will override that implied by the dateTime format property.

Examples

The following table shows a few examples of dateTime formats:

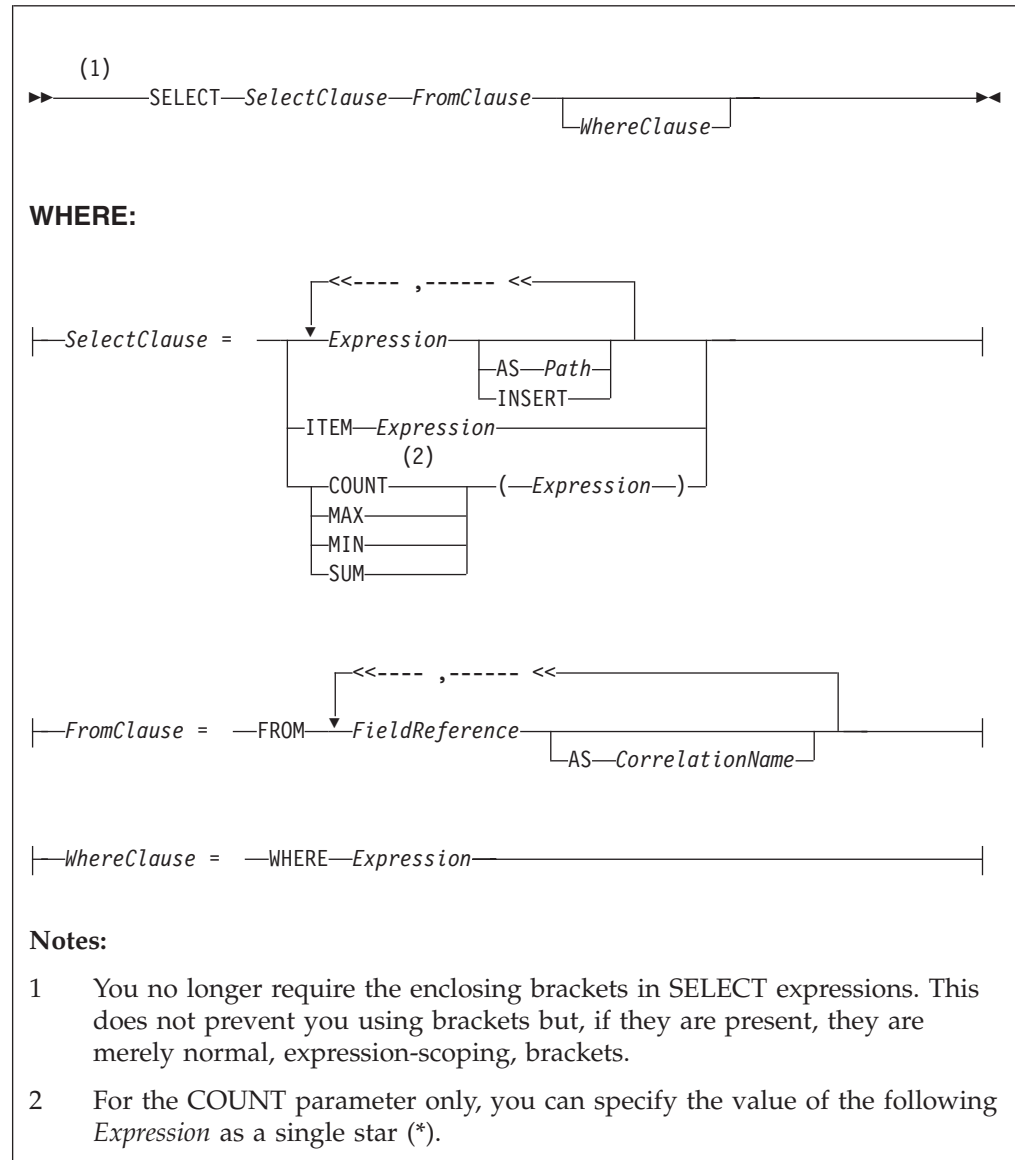
Format pattern	Result
"yyyy.MM.dd'at'HH:mm:ss ZZZ"	1996.07.10 at 15:08:56 -05:00

Format pattern	Result
EEE, MMM d, "yy"	Wed, July 10, '96
"h:mm a"	8:08 PM
"hh 'o'clock' a, ZZZZ"	09 o'clock AM, GMT+09:00
"K:mm a, ZZZ"	9:34 AM, -05:00
"yyyy.MMMMMM.dd hh:mm aaa"	1996.July.10 12:08 PM

SELECT function

The SELECT function combines, filters, and transforms complex message and database data.

Syntax



Usage

The SELECT function is the usual and most efficient way of transforming messages. You can use SELECT to:

- Comprehensively reformat messages
- Access database tables
- Make an output array that is a subset of an input array
- Make an output array that contains only the values of an input array
- Count the number of entries in an array
- Select the minimum or maximum value from a number of entries in an array
- Sum the values in an array

Introduction to SELECT

The SELECT function considers a message tree (or sub-tree) to consist of a number of “rows” and “columns”, rather like a database table. A *FieldReference* in a FROM clause identifies a field in a message tree and:

- The identified field is regarded as a “row” in a table.
- The field’s siblings are regarded as other “rows” of the same “table”.
- The field’s children are regarded as the table’s “columns”.

Note: The *FieldReference* in a FROM clause can also be a table reference that refers directly to a real database table.

The return value of the SELECT function is typically another message tree that contains “rows” whose structure and content is determined by the *SelectClause*. The number of rows in the result is the sum of all the “rows” pointed to by all the field references and table references in the FROM clause, filtered by the WHERE clause; only those fields for which the WHERE clause evaluates to TRUE are included.

The return value of the SELECT function can also be scalar (see “ITEM selections” on page 958).

You can specify the *SelectClause* in several ways; see:

- “Simple selections”
- “INSERT selections” on page 958
- “ITEM selections” on page 958
- “Column function selections” on page 958

Simple selections

To understand the SELECT function in more detail, first consider a simple case in which:

- The *SelectClause* consists of a number of expressions, each with an *AS Path* clause.
- The FROM clause contains a single *FieldReference* and an *AS CorrelationName* clause.

The SELECT function creates a local, reference, correlation variable, whose name is given by the *AS CorrelationName* clause, and then steps, in turn, through each “row” of the list of rows derived from the FROM clause. For each “row”:

1. The correlation variable is set to point to the current “row”.
2. The WHERE clause (if present) is evaluated. If it evaluates to FALSE or unknown (null), nothing is added to the result tree and processing proceeds to the next “row” of the input. Otherwise processing proceeds to the next step.
3. A new member is added to the result list.
4. The SELECT clause expressions are evaluated and assigned to fields named as dictated by the *AS Path* clause. These fields are child fields of the new member of the result list.

Typically, both the *SelectClause* and the WHERE clause expressions use the correlation variable to access “column” values (that is, fields in the input message tree) and thus to build a new message tree containing data from the input message. The correlation variable is referred to by the name specified in the *AS CorrelationName* clause or, if an *AS* clause is not specified, by the final name in the FROM *FieldReference* (that is, the name after the last dot).

Note that:

- Despite the analogy with a table, you are not restricted to accessing or creating messages with a flat, table-like, structure; you can access and build trees with arbitrarily deep folder structures.
- You are not restricted to a “column” being a single value; a column can be a repeating list value or a structure.

These concepts are best understood by reference to the examples.

If the field reference is actually a *TableReference*, the operation is very similar. In this case, the input is a real database table and is thus restricted to the flat structures supported by databases. The result tree is still not so restricted, however.

If there is more than one field reference in the FROM clause, the rightmost reference steps through each of its rows for each row in the next-to-rightmost reference, and so on. The total number of rows in the result is thus the product of the number of rows in each table. Such selects are known as *joins* and commonly use a WHERE clause that excludes most of these rows from the result. Joins are commonly used to add database data to messages.

The AS *Path* clause is optional. If it is unspecified, the broker generates a default name according to the following rules:

1. If the *SelectClause* expression is a reference to a field or a cast of a reference to a field, the name of the field is used.
2. Otherwise the broker uses the default names “Column1”, “Column2”, and so on.

Examples

The following example performs a SELECT on the table “Parts” in the schema “Shop” in the database “DSN1”. Because there is no WHERE clause, all rows are selected. Because the select clause expressions (for example, P.PartNumber) contain no AS clauses, the fields in the result adopt the same names:

```
SET PartsTable.Part[] = SELECT
  P.PartNumber,
  P.Description,
  P.Price
FROM Database.DSN1.Shop.Parts AS P;
```

If the target of the SET statement (“PartsTable”) is a variable of type ROW, after the statement is executed PartsTable will have, as children of its root element, a field called “Part” for each row in the table. Each of the “Part” fields will have child fields called “PartNumber”, “Description”, and “Price”. The child fields will have values dictated by the contents of the table. (“PartsTable” could also be a reference into a message tree).

The next example performs a similar SELECT. This case differs from the last in that the SELECT is performed on the message tree produced by the first example (rather than on a real database table). The result is assigned into a subfolder of “OutputRoot”:

```
SET OutputRoot.XML.Data.TableData.Part[] = SELECT
  P.PartNumber,
  P.Description,
  P.Price
FROM PartsTable.Part[] AS P;
```

INSERT selections

The `INSERT` clause is an alternative to the `AS` clause. It assigns the result of the *SelectClause* expression (which must be a row) to the current new row itself, rather than to a child of it. The effect of this is to merge the row result of the expression into the row being generated by the `SELECT`. This differs from the `AS` clause, in that the `AS` clause always generates at least one child element before adding a result, whereas `INSERT` generates none. `INSERT` is useful when inserting data from other `SELECT` operations, because it allows the data to be merged without extra folders.

ITEM selections

The *SelectClause* can consist of the keyword `ITEM` and a single expression. The effect of this is to make the results nameless. That is, the result is a list of values of the type returned by the expression, rather than a row. This option has several uses:

- In conjunction with a scalar expression and the `THE` function, it can be used to create a `SELECT` query that returns a single scalar value (for example, the price of a particular item from a table).
- In conjunction with a `CASE` expression and `ROW` constructors, it can be used to create a `SELECT` query that creates or handles messages in which the structure of some “rows” (that is, repeats in the message) is different to others. This is useful for handling messages that have a repeating structure but in which the repeats do not all have the same structure.
- In conjunction with a `ROW` constructor, it can be used to create a `SELECT` query that collapses levels of repetition in the input message.

Column function selections

The *SelectClause* can consist of one of the functions `COUNT`, `MAX`, `MIN`, and `SUM` operating on an expression. These functions are known as column functions. They return a single scalar value (not a list) giving the count, maximum, minimum, or sum of the values that *Expression* evaluated to in stepping through the rows of the `FROM` clause. If *Expression* evaluates to `NULL` for a particular row, the value is ignored, so that the function returns the count, maximum, minimum, or sum of the remaining rows.

For the `COUNT` function only, *Expression* can consist of a single star (*). This form counts the rows regardless of null values.

To make the result a useful reflection of the input message, *Expression* typically includes the correlation variable.

Typically, *Expression* evaluates to the same data type for each row. In these cases, the result of the `MAX`, `MIN`, and `SUM` functions will be of the same data type as the operands. The returned values are not required to be all of the same type, however, and, if they are not, the normal rules of arithmetic apply. For example, if a field in a repeated message structure contains integer values for some rows and float values for others, the sum follows the normal rules for addition. It would be of type float because the operation is equivalent to adding a number of integer and float values.

The result of the `COUNT` function is always an integer.

Differences between message and database selections

FROM expressions in which a correlation variable represents a row in a message behave slightly differently from those in which the correlation variable represents a row in a real database table.

In the message case, a path involving a star (*) has the normal meaning; it ignores the field's name and finds the first field that matches the other criteria (if any).

In the database case a star (*) has, for historical reasons, the special meaning of "all fields". This special meaning requires advance knowledge of the definition of the database table and is only supported when querying the default database (that is, the database pointed to by the node's data source attribute). For example, the following queries return column name/value pairs only when querying the default database:

```
SELECT * FROM Database.Datasource.SchemaName.Table As A
SELECT A.* FROM Database.Datasource.SchemaName.Table As A
SELECT A FROM Database.Datasource.SchemaName.Table AS A
```

Specifying the SELECT expressions

SelectClause

SelectClause expressions can use any of the broker's operators and functions in any combination. They can refer to the tables' columns, message fields, correlation names declared by containing SELECTs, and to any other declared variables or constants that are in scope.

AS Path

An *AS Path* expression is a relative path (that is, there is no correlation name) but is otherwise unrestricted in any way. For example, it can contain:

- Indices (for example, A.B.C[i])
- Field-type specifiers (for example, A.B.(XML.Attribute)C)
- Multipart paths (for example, A.B.C)
- Name expressions (for example, A.B.{var})

Any expressions in these paths can also use any of the broker's operators and functions in any combination. The expressions can refer to the tables' columns, message fields, correlation names declared by containing SELECTs, and any declared variables or constants.

FROM clause

FROM clause expressions can contain multiple database references, multiple message references, or a mixture of the two. You can join tables with tables, messages with messages, or tables with messages.

FROM clause *FieldReferences* can contain expressions of any kind (for example, Database.{DataSource}.{Schema}.Table1).

You can calculate a field, data source, schema, or table name at run time.

WHERE clause

The WHERE clause expression can use any of the broker's operators and functions in any combination. It can refer to table columns, message fields, and any declared variables or constants.

However, be aware that the broker treats the WHERE clause expression by examining the expression and deciding whether the whole expression can be

evaluated by the database. If it can, it is given to the database. In order to be evaluated by the database, it must use only those functions and operators supported by the database.

The WHERE clause can, however, refer to message fields, correlation names declared by containing SELECTs, and to any other declared variables or constants within scope.

If the whole expression cannot be evaluated by the database, the broker looks for top-level AND operators and examines each sub-expression separately. It then attempts to give the database those sub-expressions that it can evaluate, leaving the broker to evaluate the rest. You need to be aware of this situation for two reasons:

1. Apparently trivial changes to WHERE clause expressions can have large effects on performance. You can determine how much of the expression was given to the database by examining a user trace.
2. Some databases' functions exhibit subtle differences of behavior from those of the broker.

Relation to the THE function

You can use the function THE (which returns the first element of a list) in conjunction with SELECT to produce a non-list result. This is useful, for example, when a SELECT query is required to return no more than one item. It is particularly useful in conjunction with ITEM (see "ITEM selections" on page 958).

Differences from the SQL standard

ESQL SELECT differs from database SQL SELECT in the following ways:

- ESQL can produce tree-structured result data
- ESQL can accept arrays in SELECT clauses
- ESQL has the THE function and the ITEM and INSERT parameters
- ESQL has no SELECT ALL function in this release
- ESQL has no ORDER BY function in this release
- ESQL has no SELECT DISTINCT function in this release
- ESQL has no GROUP BY or HAVING parameters in this release
- ESQL has no AVG column function in this release

Restrictions

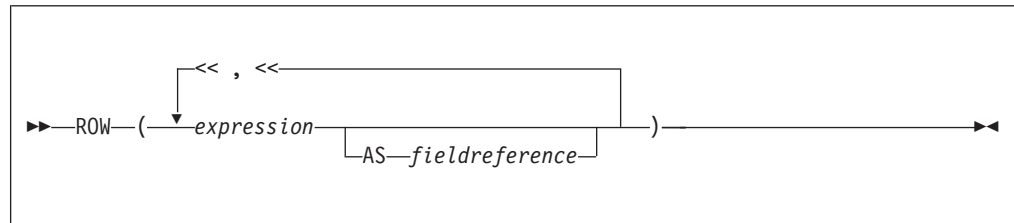
The following restrictions apply to the current release:

- When a SELECT command operates on more than one database table, all the tables must be in the same database instance. (That is, the *TableReferences* must not specify different data source names.)
- If the FROM clause refers to both messages and tables, the tables must precede the messages in the list.

ROW constructor function

ROW constructor is a complex function used to explicitly generate rows of values that can be assigned to fields in an output message.

Syntax



A ROW consists of a sequence of named values. When assigned to a field reference it creates that sequence of named values as child fields of the referenced field. A ROW cannot be assigned to an array field reference.

Examples:

Example 1

```
SET OutputRoot.XML.Data = ROW('granary' AS bread,
                              'riesling' AS wine,
                              'stilton' AS cheese);
```

produces:

```
<Data>
  <bread>granary</bread>
  <wine>riesling</wine>
  <cheese>stilton</cheese>
</Data>
```

Example 2

Given the following XML input message body:

```
<Proof>
  <beer>5</beer>
  <wine>12</wine>
  <gin>40</gin>
</Proof>
```

the following ESQL:

```
SET OutputRoot.XML.Data = ROW(InputBody.Proof.beer,
                              InputBody.Proof.wine AS vin,
                              (InputBody.Proof.gin * 2) AS special);
```

produces the following result:

```
<Data>
  <beer>5</beer>
  <vin>12</vin>
  <special>80</special>
</Data>
```

Because the values in this case are derived from field references that already have names, it is not necessary to explicitly provide a name for each element of the row, but you might choose to do so.

LIST constructor function

The LIST constructor complex function is used to explicitly generate lists of values that can be assigned to fields in an output message.

Syntax



A LIST consists of a sequence of unnamed values. When assigned to an array field reference (indicated by [] suffixed to the last element of the reference), each value is assigned in sequence to an element of the array. A LIST cannot be assigned to a non-array field reference.

Examples:

Example 1

Given the following XML message input body:

```
<Car>
  <size>big</size>
  <color>red</color>
</Car>
```

The following ESQL:

```
SET OutputRoot.XML.Data.Result[] = LIST{InputBody.Car.colour,
                                          'green',
                                          'blue'};
```

produces the following results:

```
<Data>
  <Result>red</Result>
  <Result>green</Result>
  <Result>blue</Result>
</Data>
```

In the case of a LIST, there is no explicit name associated with each value. The values are assigned in sequence to elements of the message field array specified as the target of the assignment. Curly braces rather than parentheses are used to surround the LIST items.

Example 2

Given the following XML input message body:

```
<Data>
  <Field>Keats</Field>
  <Field>Shelley</Field>
  <Field>Wordsworth</Field>
  <Field>Tennyson</Field>
  <Field>Byron</Field>
</Data>
```

the following ESQL:

```
-- Copy the entire input message to the output message,
-- including the XML message field array as above
SET OutputRoot = InputRoot;
SET OutputRoot.XML.Data.Field[] = LIST{'Henri','McGough','Patten'};
```

Produces the following output:

```

<Data>
  <Field>Henri</Field>
  <Field>McGough</Field>
  <Field>Patten</Field>
</Data>

```

The previous members of the `Data.Field[]` array have been discarded. Assigning a new list of values to an already existing message field array removes all the elements in the existing array before the new ones are assigned.

ROW and LIST combined

ROW and LIST combined form a complex function.

A ROW might validly be an element in a LIST. For example:

```

SET OutputRoot.XML.Data.Country[] =
  LIST{ROW('UK' AS name, 'pound' AS currency),
        ROW('US' AS name, 'dollar' AS currency),
        'default'};

```

produces the following result:

```

<Data>
  <Country>
    <name>UK</name>
    <currency>pound</currency>
  </Country>
  <Country>
    <name>US</name>
    <currency>dollar</currency>
  </Country>
  <Country>default</Country>
</Data>

```

ROW and non-ROW values can be freely mixed within a LIST.

A LIST cannot be a member of a ROW. Only named scalar values can be members of a ROW.

ROW and LIST comparisons

You can compare ROWs and LISTs against other ROWs and LISTs.

Examples:

Example 1

```

IF ROW(InputBody.Data.*[1],InputBody.Data.*[2]) =
  ROW('Raf' AS Name,'25' AS Age) THEN ...
IF LIST{InputBody.Data.Name, InputBody.Data.Age} = LIST{'Raf','25'} THEN ...

```

With the following XML input message body both the IF expressions in both the above statements evaluate to TRUE:

```

<Data>
  <Name>Raf</Name>
  <Age>25</Age>
</Data>

```

In the comparison between ROWs, both the name and the value of each element are compared; in the comparison between LISTs only the value of each element is compared. In both cases, the cardinality and sequential order of the LIST or ROW operands being compared must be equal in order for the two operands to be equal.

In other words, all the following are false because either the sequential order or the cardinality of the operands being compared do not match:

```
ROW('alpha' AS A, 'beta' AS B) =  
    ROW('alpha' AS A, 'beta' AS B, 'delta' AS D)  
ROW('alpha' AS A, 'beta' AS B) =  
    ROW('beta' AS B, 'alpha' AS A)  
LIST{1,2,3} = LIST{1,2,3,4}  
LIST{3,2,1} = LIST{1,2,3}
```

Example 2

Consider the following ESQL:

```
IF InputBody.Places =  
    ROW('Ken' AS first, 'Bob' AS second, 'Kate' AS third) THEN ...
```

With the following XML input message body, the above IF expression evaluates to TRUE:

```
<Places>  
  <first>Ken</first>  
  <second>Bob</second>  
  <third>Kate</third>  
</Places>
```

The presence of an explicitly-constructed ROW as one of the operands to the comparison operator results in the other operand also being treated as a ROW.

Contrast this with a comparison such as:

```
IF InputBody.Lottery.FirstDraw = InputBody.Lottery.SecondDraw THEN ...
```

which compares the value of the FirstDraw and SecondDraw fields, not the names and values of each of FirstDraw and SecondDraw's child fields constructed as a ROW. Thus an XML input message body such as:

```
<Lottery>  
  <FirstDraw>wednesday  
    <ball1>32</ball1>  
    <ball2>12</ball2>  
  </FirstDraw>  
  <SecondDraw>saturday  
    <ball1>32</ball1>  
    <ball2>12</ball2>  
  </SecondDraw>  
</Lottery>
```

would not result in the above IF expression being evaluated as TRUE, because the values wednesday and saturday are being compared, not the names and values of the ball fields.

Example 3

Consider the following ESQL:

```
IF InputBody.Cities.City[] = LIST{'Athens','Sparta','Thebes'} THEN ...
```

With the following XML input message body, the IF expression evaluates to TRUE:

```
<Cities>  
  <City>Athens</City>  
  <City>Sparta</City>  
  <City>Thebes</City>  
</Cities>
```

Two message field arrays can be compared together in this way, for example:

```
IF InputBody.Cities.Mediaeval.City[] =
    InputBody.Cities.Modern.City[] THEN ...

IF InputBody.Cities.Mediaeval.*[] = InputBody.Cities.Modern.*[] THEN ...

IF InputBody.Cities.Mediaeval.(XML.Element)[] =
    InputBody.Cities.Modern.(XML.Element)[] THEN ...
```

With the following XML input message body, the IF expression of the first and third of the statements above evaluates to TRUE:

```
<Cities>
  <Mediaeval>1350
    <City>London</City>
    <City>Paris</City>
  </Mediaeval>
  <Modern>1990
    <City>London</City>
    <City>Paris</City>
  </Modern>
</Cities>
```

However the IF expression of the second statement evaluates to FALSE, because the *[] indicates that all the children of Mediaeval and Modern are to be compared, not just the (XML.Element)s. In this case the values 1350 and 1990, which form nameless children of Mediaeval and Modern, are compared as well as the values of the City tags.

The IF expression of the third statement above evaluates to TRUE with an XML input message body such as:

```
<Cities>
  <Mediaeval>1350
    <Location>London</Location>
    <Location>Paris</Location>
  </Mediaeval>
  <Modern>1990
    <City>London</City>
    <City>Paris</City>
  </Modern>
</Cities>
```

LISTs are composed of unnamed values. It is the values of the child fields of Mediaeval and Modern that are compared, not their names.

Supported casts

This topic lists the CASTs that are supported between combinations of data-types.

A CAST is not supported between every combination of data-types. Those that are supported are listed below, along with the effect of the CAST.

When casting, there can be a one-to-one or a many-to-one mapping between the source data-type and the target data-type. An example of a one-to-one mapping is where the source data-type is a single integer and the target data-type a single float. An example of a many-to-one mapping is where the source data consists of three integers that are converted to a single date. Table 16 on page 966 lists the supported one-to-one casts. Table 17 on page 972 lists the supported many-to-one casts.

See “ESQL data types” on page 147 for information about precision, scale, and interval qualifier.

Table 16. Supported casts: one-to-one mappings of source to target data-type

Source data-type	Target data-type	Effect
BIT	BIT	The result is the same as the input.
BIT	BLOB	The bit array is converted to a byte array with a maximum of 2^{63} elements. An error is reported if the source is not of a suitable length to produce a BLOB (that is a multiple of 8).
BIT	CHARACTER	<p>The result is a string conforming to the definition of a bit string literal whose interpreted value is the same as the source value. The resulting string has the form B'bbbbbb' (where b is either 0 or 1).</p> <p>If you specify either a CCSID or ENCODING clause, the bit array is assumed to be characters in the specified CCSID and encoding, and is code-page converted into the character return value.</p> <p>If you specify only a CCSID, big endian encoding is assumed.</p> <p>If you specify only an encoding, a CCSID of 1208 is assumed.</p> <p>This function reports conversion errors if the code page or encoding are unknown, the data supplied is not an integral number of characters of the code page, or the data contains characters that are not valid in the given code page.</p>
BIT	INTEGER	The bit array has a maximum of 2^{63} elements and is converted to an integer. An error is reported if the source is not of the correct length to match an integer.
BLOB	BIT	The given byte array is converted to a bit array with a maximum of 2^{63} elements.
BLOB	BLOB	The result is the same as the input.
BLOB	CHARACTER	<p>The result is a string conforming to the definition of a binary string literal whose interpreted value is the same as the source value. The resulting string has the form X'hhhh' (where h is any hexadecimal character).</p> <p>If you specify either a CCSID or ENCODING clause, the byte array is assumed to be characters in the specified CCSID and encoding, and is code-page converted into the character return value.</p> <p>If you specify only a CCSID, big endian encoding is assumed.</p> <p>If you specify only an encoding, a CCSID of 1208 is assumed.</p> <p>This function reports conversion errors if the code page or encoding are unknown, the data supplied is not an integral number of characters of the code page, or the data contains characters that are not valid in the given code page.</p>
BLOB	INTEGER	The byte array has a maximum of 2^{63} elements and is converted to an integer. An error is reported if the source is not of the correct length to match an integer.
BOOLEAN	BOOLEAN	The result is the same as the input.
BOOLEAN	CHARACTER	If the source value is TRUE, the result is the character string TRUE. If the source value is FALSE, the result is the character string FALSE. Because the UNKNOWN Boolean value is the same as the NULL value for Booleans, the result is NULL if the source value is UNKNOWN.

Table 16. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
CHARACTER	BIT	<p>The character string must conform to the rules for a bit string literal or for the contents of the bit string literal. That is, the character string can be of the form B'bbbbbbb' or bbbbbbb (where b' can be either 0 or 1).</p> <p>If you specify either a CCSID or ENCODING clause, the character string is converted into the specified CCSID and encoding and placed without further conversion into the bit array return value.</p> <p>If you specify only a CCSID, big endian encoding is assumed.</p> <p>If you specify only an encoding, a CCSID of 1208 is assumed.</p> <p>This function reports conversion errors if the code page or encoding are unknown or the data contains Unicode characters that cannot be converted to the given code page.</p>
CHARACTER	BLOB	<p>The character string must conform to the rules for a binary string literal or for the contents of the binary string literal. That is, the character string can be of the form X'hhhhhh' or hhhhhh (where h can be any hexadecimal characters).</p> <p>If you specify either a CCSID or ENCODING clause, the character string is converted into the specified CCSID and encoding and placed without further conversion into the byte array return value.</p> <p>If you specify only a CCSID, big endian encoding is assumed.</p> <p>If you specify only an encoding, a CCSID of 1208 is assumed.</p> <p>This function reports conversion errors if the code page or encoding are unknown or the data contains Unicode characters that cannot be converted to the given code page.</p>
CHARACTER	BOOLEAN	<p>The character string is interpreted in the same way as a Boolean literal. That is, the character string must be one of the strings TRUE, FALSE, or UNKNOWN (in any case combination).</p>
CHARACTER	CHARACTER	<p>The result is the same as the input.</p>
CHARACTER	DATE	<p>If a FORMAT clause is not specified, the character string must conform to the rules for a date literal or the date string. That is, the character string can be either DATE '2002-10-05' or 2002-10-05.</p> <p>See also "Formatting and parsing dateTimes as strings" on page 948.</p>
CHARACTER	DECIMAL	<p>The character string is interpreted in the same way as an exact numeric literal to form a temporary decimal result with a scale and precision defined by the format of the string. This is converted into a decimal of the specified precision and scale, with a runtime error being generated if the conversion results in loss of significant digits.</p> <p>If you do not specify the precision and scale, the precision and scale of the result are the minimum necessary to hold the given value.</p> <p>The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing numbers as strings" on page 945.</p>
CHARACTER	FLOAT	<p>The character string is interpreted in the same way as a floating point literal.</p> <p>The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing numbers as strings" on page 945.</p>

Table 16. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
CHARACTER	GMTTIME	The character string must conform to the rules for a GMT time literal or the time string. That is, the character string can be either GMTTIME '09:24:15' or 09:24:15. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 948.
CHARACTER	GMTTIMESTAMP	The character string must conform to the rules for a GMT timestamp literal or the timestamp string. That is, the character string can be either GMTTIMESTAMP '2002-10-05 09:24:15' or 2002-10-05 09:24:15. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 948.
CHARACTER	INTEGER	The character string is interpreted in the same way as an integer literal. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing numbers as strings" on page 945.
CHARACTER	INTERVAL	The character string must conform to the rules for an interval literal with the same interval qualifier as specified in the CAST function, or it must conform to the rules for an interval string that apply for the specified interval qualifier.
CHARACTER	TIME	The character string must conform to the rules for a time literal or for the time string. That is, the character string can be either TIME '09:24:15' or 09:24:15. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 948.
CHARACTER	TIMESTAMP	The character string must conform to the rules for a timestamp literal or for the timestamp string. That is, the character string can be either TIMESTAMP '2002-10-05 09:24:15' or 2002-10-05 09:24:15. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 948.
DATE	CHARACTER	The result is a string conforming to the definition of a date literal, whose interpreted value is the same as the source date value. For example: CAST (DATE '2002-10-05' AS CHARACTER) returns DATE '2002-10-05' The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 948.
DATE	DATE	The result is the same as the input.
DATE	GMTTIMESTAMP	The result is a value whose date fields are taken from the source date value, and whose time fields are taken from the current GMT time.
DATE	TIMESTAMP	The result is a value whose date fields are taken from the source date value, and whose time fields are taken from the current time.
DECIMAL	CHARACTER	The result is the shortest character string that conforms to the definition of an exact numeric literal and whose interpreted value is the value of the decimal. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing numbers as strings" on page 945.

Table 16. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
DECIMAL	DECIMAL	The value is converted to the specified precision and scale, with a runtime error being generated if the conversion results in loss of significant digits. If you do not specify the precision and scale, the value, precision and scale are preserved; it is a NOOP (no operation).
DECIMAL	FLOAT	The number is converted, with rounding if necessary.
DECIMAL	INTEGER	The value is rounded and converted into an integer, with a runtime error being generated if the conversion results in loss of significant digits.
DECIMAL	INTERVAL	If the interval qualifier specified has only one field, the result is an interval with that qualifier with the field equal to the value of the exact numeric. Otherwise a runtime error is generated.
FLOAT	CHARACTER	The result is the shortest character string that conforms to the definition of an approximate numeric literal and whose mantissa consists of a single digit that is not 0, followed by a period and an unsigned integer, and whose interpreted value is the value of the float. The behavior changes if the FORMAT clause is specified. See also “Formatting and parsing numbers as strings” on page 945.
FLOAT	FLOAT	The result is the same as the input.
FLOAT	DECIMAL	The value is rounded and converted into a decimal of the specified precision and scale, with a runtime error being generated if the conversion results in loss of significant digits. If you do not specify the precision and scale, the precision and scale of the result are the minimum necessary to hold the given value.
FLOAT	INTEGER	The value is rounded and converted into an integer, with a runtime error being generated if the conversion results in loss of significant digits.
FLOAT	INTERVAL	If the specified interval qualifier has only one field, the result is an interval with that qualifier with the field equal to the value of the numeric. Otherwise a runtime error is generated.
GMTTIME	CHARACTER	The result is a string conforming to the definition of a GMTTIME literal whose interpreted value is the same as the source value. The resulting string has the form GMTTIME ‘hh:mm:ss’. The behavior changes if the FORMAT clause is specified. See also “Formatting and parsing dateTimes as strings” on page 948.
GMTTIME	GMTTIME	The result is the same as the input.
GMTTIME	TIME	The resulting value is the source value plus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.
GMTTIME	GMTTIMESTAMP	The result is a value whose date fields are taken from the current date, and whose time fields are taken from the source GMT time.
GMTTIME	TIMESTAMP	The result is a value whose date fields are taken from the current date, and whose time fields are taken from the source GMT time, plus the local time zone displacement (as returned by LOCAL_TIMEZONE).

Table 16. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
GMTTIMESTAMP	CHARACTER	The result is a string conforming to the definition of a GMTTIMESTAMP literal whose interpreted value is the same as the source value. The resulting string has the form GMTTIMESTAMP 'yyyy-mm-dd hh:mm:ss'. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 948.
GMTTIMESTAMP	DATE	The result is a value whose fields consist of the date fields of the source GMTTIMESTAMP value.
GMTTIMESTAMP	GMTTIME	The result is a value whose fields consist of the time fields of the source GMTTIMESTAMP value.
GMTTIMESTAMP	TIME	The result is a value whose time fields are taken from the source GMTTIMESTAMP value, plus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.
GMTTIMESTAMP	GMTTIMESTAMP	The result is the same as the input.
GMTTIMESTAMP	TIMESTAMP	The resulting value is source value plus the local time zone displacement (as returned by LOCAL_TIMEZONE).
INTEGER	BIT	The given integer is converted to a bit array with a maximum of 2^{63} elements.
INTEGER	BLOB	The given integer is converted to a byte array with a maximum of 2^{63} elements.
INTEGER	CHARACTER	The result is the shortest character string that conforms to the definition of an exact numeric literal and whose interpreted value is the value of the integer. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing numbers as strings" on page 945.
INTEGER	FLOAT	The number is converted, with rounding if necessary.
INTEGER	INTEGER	The result is the same as the input.
INTEGER	DECIMAL	The value is converted into a decimal of the specified precision and scale, with a runtime error being generated if the conversion results in loss of significant digits. If you do not specify the precision and scale, the precision and scale of the result are the minimum necessary to hold the given value.
INTEGER	INTERVAL	If the interval qualifier specified has only one field, the result is an interval with that qualifier with the field equal to the value of the exact numeric. Otherwise a runtime error is generated.
INTERVAL	CHARACTER	The result is a string conforming to the definition of an INTERVAL literal, whose interpreted value is the same as the source interval value. For example: CAST(INTERVAL '4' YEARS AS CHARACTER) returns INTERVAL '4' YEARS

Table 16. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
INTERVAL	DECIMAL	If the interval value has a qualifier that has only one field, the result is a decimal of the specified precision and scale with that value, with a runtime error being generated if the conversion results in loss of significant digits. If the interval has a qualifier with more than one field, such as YEAR TO MONTH, a runtime error is generated. If you do not specify the precision and scale, the precision and scale of the result are the minimum necessary to hold the given value.
INTERVAL	FLOAT	If the interval value has a qualifier that has only one field, the result is a float with that value. If the interval has a qualifier with more than one field, such as YEAR TO MONTH, a runtime error is generated.
INTERVAL	INTEGER	If the interval value has a qualifier that has only one field, the result is an integer with that value. If the interval has a qualifier with more than one field, such as YEAR TO MONTH, a runtime error is generated.
INTERVAL	INTERVAL	The result is the same as the input. Year-month intervals can be converted only to year-month intervals, and day-second intervals only to day-second intervals. The source interval is converted into a scalar in units of the least significant field of the target interval qualifier. This value is normalized into an interval with the target interval qualifier. For example, to convert an interval that has the qualifier MINUTE TO SECOND into an interval with the qualifier DAY TO HOUR, the source value is converted into a scalar in units of hours, and this value is normalized into an interval with qualifier DAY TO HOUR.
TIME	CHARACTER	The result is a string conforming to the definition of a TIME literal, whose interpreted value is the same as the source time value. For example: <code>CAST(TIME '09:24:15' AS CHARACTER)</code> returns <code>TIME '09:24:15'</code> The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 948.
TIME	GMTTIME	The result value is the source value minus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.
TIME	GMTTIMESTAMP	The result is a value whose date fields are taken from the current date, and whose time fields are taken from the source GMT time, minus the local time zone displacement (as returned by LOCAL_TIMEZONE).
TIME	TIME	The result is the same as the input.
TIME	TIMESTAMP	The result is a value whose date fields are taken from the current date, and whose time fields are taken from the source time value.

Table 16. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
TIMESTAMP	CHARACTER	The result is a string conforming to the definition of a <code>TIMESTAMP</code> literal, whose interpreted value is the same as the source timestamp value. For example: <code>CAST(TIMESTAMP '2002-10-05 09:24:15' AS CHARACTER)</code> returns <code>TIMESTAMP '2002-10-05 09:24:15'</code> The behavior changes if the <code>FORMAT</code> clause is specified. See also “Formatting and parsing dateTimes as strings” on page 948.
TIMESTAMP	DATE	The result is a value whose fields consist of the date fields of the source timestamp value.
TIMESTAMP	GMTTIME	The result is a value whose time fields are taken from the source <code>TIMESTAMP</code> value, minus the local time zone displacement (as returned by <code>LOCAL_TIMEZONE</code>). The hours field is calculated modulo 24.
TIMESTAMP	GMTTIMESTAMP	The resulting value is the source value minus the local time zone displacement (as returned by <code>LOCAL_TIMEZONE</code>).
TIMESTAMP	TIME	The result is a value whose fields consist of the time fields of the source timestamp value.
TIMESTAMP	TIMESTAMP	The result is the same as the input.

Table 17. Supported casts: many-to-one mappings of source to target data-type

Source data-type	Target data-type	Effect
Numeric, Numeric, Numeric	DATE	Creates a <code>DATE</code> value from the numerics in the order year, month, and day. Non-integer values are rounded.
Numeric, Numeric, Numeric	TIME	Creates a <code>TIME</code> value from the numerics in the order hours, minutes, and seconds. Non-integer values for hours and minutes are rounded.
Numeric, Numeric, Numeric	GMTTIME	Creates a <code>GMTTIME</code> value from the numerics in the order of hours, minutes, and seconds. Non-integer values for hours and minutes are rounded.
Numeric, Numeric, Numeric, Numeric, Numeric, Numeric	TIMESTAMP	Creates a <code>TIMESTAMP</code> value from the numerics in the order years, months, days, hours, minutes, and seconds. Non-integer values for years, months, days, hours, and minutes are rounded.
Numeric, Numeric, Numeric, Numeric, Numeric, Numeric	GMTTIMESTAMP	Creates a <code>GMTTIMESTAMP</code> value from the numerics in the order years, months, days, hours, minutes, and seconds. Non-integer values for years, months, days, hours, and minutes are rounded.
DATE, TIME	TIMESTAMP	The result is a <code>TIMESTAMP</code> value with the given <code>DATE</code> and <code>TIME</code> .
DATE, GMTTIME	GMTTIMESTAMP	The result is a <code>GMTTIMESTAMP</code> value with the given <code>DATE</code> and <code>GMTTIME</code> .
Numeric, Numeric	INTERVAL YEAR TO MONTH	The result is an <code>INTERVAL</code> with the first source as years and the second as months. Non-integer values are rounded.
Numeric, Numeric	INTERVAL HOUR TO MINUTE	The result is an <code>INTERVAL</code> with the first source as hours and the second as minutes. Non-integer values are rounded.
Numeric, Numeric, Numeric	INTERVAL HOUR TO SECOND	The result is an <code>INTERVAL</code> with the sources as hours, minutes, and seconds, respectively. Non-integer values for hours and minutes are rounded.

Table 17. Supported casts: many-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
Numeric, Numeric	INTERVAL MINUTE TO SECOND	The result is an INTERVAL with the sources as minutes and seconds, respectively. Non-integer values for minutes are rounded.
Numeric, Numeric	INTERVAL DAY TO HOUR	The result is an INTERVAL with the sources as days and hours, respectively. Non-integer values are rounded.
Numeric, Numeric, Numeric	INTERVAL DAY TO MINUTE	The result is an INTERVAL with the sources as days, hours, and minutes, respectively. Non-integer values are rounded.
Numeric, Numeric, Numeric, Numeric	INTERVAL DAY TO SECOND	The result is an INTERVAL with the sources as days, hours, minutes, and seconds, respectively. Non-integer values for days, hours, and minutes are rounded.
Numeric	INTERVAL YEAR	The result is an INTERVAL with the source as years, rounded if necessary.
Numeric	INTERVAL MONTH	The result is an INTERVAL with the source as months, rounded if necessary.
Numeric	INTERVAL DAY	The result is an INTERVAL with the source as days, rounded if necessary.
Numeric	INTERVAL HOUR	The result is an INTERVAL with the source as hours, rounded if necessary.
Numeric	INTERVAL MINUTE	The result is an INTERVAL with the source as minutes, rounded if necessary.
Numeric	INTERVAL SECOND	The result is an INTERVAL with the source as seconds.

Implicit casts

This topic discusses implicit casts.

It is not always necessary to cast values between types. Some casts are done implicitly. For example, numbers are implicitly cast between the three numeric types for the purposes of comparison and arithmetic. Character strings are also implicitly cast to other data types for the purposes of comparison.

There are three situations in which a data value of one type is cast to another type implicitly. The behavior and restrictions of the implicit cast are the same as described for the explicit cast function, except where noted in the topics listed below.

Implicit CASTs for comparisons

The standard SQL comparison operators $>$, $<$, $>=$, $<=$, $=$, $<>$ are supported for comparing two values in ESQL.

When the data types of the two values are not the same, one of them can be implicitly cast to the type of the other to allow the comparison to proceed. In the table below, the vertical axis represents the left hand operand, the horizontal axis represents the right hand operand.

L means that the right hand operand is cast to the type of the left hand operand before comparison; R means the opposite; X means that no implicit casting takes place; a blank means that comparison between the values of the two data types is not supported.

	ukn	bln	int	float	dec	char	time	gtm	date	ts	gts	ivl	blob	bit
ukn														
bln		X				L								
int			X	R	R	L								
float			L	X	L	L								
dec			L	R	X	L								
chr		R	R	R	R	X	R	R	R	R	R	R ¹	R	R
tm						L	X	L						
gtm						L	R	X						
dt						L			X	R ²	R ²			
ts						L			L ²	X	L			
gts						L			L ²	R	X			
ivl						L ¹						X		
blb						L							X	
bit						L								X

Notes:

1. When casting from a character string to an interval, the character string must be of the format INTERVAL '`<values>`' `<qualifier>`. The format `<values>`, which is allowed for an explicit CAST, is not allowed here because no qualifier external to the string is supplied.
2. When casting from a DATE to a TIMESTAMP or GMTTIMESTAMP, the time portion of the TIMESTAMP is set to all zero values (00:00:00). This is different from the behavior of the explicit cast, which sets the time portion to the current time.

Numeric types:

The comparison operators operate on all three numeric types.

Character strings:

You cannot define an alternative collation order that, for example, collates upper and lowercase characters equally.

When comparing character strings, trailing blanks are not significant, so the comparison 'hello' = 'hello ' returns true.

Datetime values:

Datetime values are compared in accordance with the natural rules of the Gregorian calendar and clock.

You can compare the time zone you are working in with the GMT time zone. The GMT time zone is converted into a local time zone based on the difference between your local time zone and the GMT time specified. When you compare your local time with the GMT time, the comparison is based on the difference at a given time on a given date.

Conversion is always based on the value of LOCAL_TIMEZONE. This is because GMT timestamps are converted to local timestamps only if it can be done unambiguously. Converting a local timestamp to a GMT timestamp has difficulties around the daylight saving cut-over time, and converting between times and GMT

times (without date information) has to be done based on the LOCAL_TIMEZONE value, because you cannot specify which time zone difference to use otherwise.

Booleans:

Boolean values can be compared using all the normal comparison operators. The TRUE value is defined to be greater than the FALSE value. Comparing either value to the UNKNOWN boolean value (which is equivalent to NULL) returns an UNKNOWN result.

Intervals:

Intervals are compared by converting the two interval values into intermediate representations, so that both intervals have the same interval qualifier. Year-month intervals can be compared only with other year-month intervals, and day-second intervals can be compared only with other day-second intervals.

For example, if an interval in minutes, such as INTERVAL '120' MINUTE is compared with an interval in days to seconds, such as INTERVAL '0 02:01:00', the two intervals are first converted into values that have consistent interval qualifiers, which can be compared. So, in this example, the first value is converted into an interval in days to seconds, which gives INTERVAL '0 02:00:00', which can be compared with the second value.

Comparing character strings with other types:

If a character string is compared with a value of another type, WebSphere Message Broker attempts to cast the character string into a value of the same data type as the other value.

For example, you can write an expression:

```
'1234' > 4567
```

The character string on the left is converted into an integer before the comparison takes place. This behavior reduces some of the need for explicit CAST operators when comparing values derived from a generic XML message with literal values. (For details of explicit casts that are supported, see “Supported casts” on page 965.) It is this facility that allows you to write the following expression:

```
Body.Trade.Quantity > 5000
```

In this example, the field reference on the left evaluates to the character string '1000' and, because this is being compared to an integer, that character string is converted into an integer before the comparison takes place.

You must still check whether the price field that you want interpreted as a decimal is greater than a given threshold. Make sure that the literal you compare it to is a decimal value and not an integer.

For example:

```
Body.Trade.Price > 100
```

does not have the desired effect, because the Price field is converted into an integer, and that conversion fails because the character string contains a decimal point. However, the following expression succeeds:

```
Body.Trade.Price > 100.00
```

Implicit CASTs for arithmetic operations

This topic lists the implicit CASTs available for arithmetic operations.

Normally the arithmetic operators (+, -, *, and /) operate on operands of the same data type, and return a value of the same data type as the operands. Cases where it is acceptable for the operands to be of different data types, or where the data type of the resulting value is different from the type of the operands, are shown in the following table.

The following table lists the implicit CASTs for arithmetic operation.

Left operand data type	Right operand data type	Supported operators	Result data type
INTEGER	FLOAT	+, -, *, /	FLOAT ¹
INTEGER	DECIMAL	+, -, *, /	DECIMAL ¹
INTEGER	INTERVAL	*	INTERVAL ⁴
FLOAT	INTEGER	+, -, *, /	FLOAT ¹
FLOAT	DECIMAL	+, -, *, /	FLOAT ¹
FLOAT	INTERVAL	*	INTERVAL ⁴
DECIMAL	INTEGER	+, -, *, /	DECIMAL ¹
DECIMAL	FLOAT	+, -, *, /	FLOAT ¹
DECIMAL	INTERVAL	*	INTERVAL ⁴
TIME	TIME	-	INTERVAL ²
TIME	GMTTIME	-	INTERVAL ²
TIME	INTERVAL	+, -	TIME ³
GMTTIME	TIME	-	INTERVAL ²
GMTTIME	GMTTIME	-	INTERVAL ²
GMTTIME	INTERVAL	+, -	GMTTIME ³
DATE	DATE	-	INTERVAL ²
DATE	INTERVAL	+, -	DATE ³
TIMESTAMP	TIMESTAMP	-	INTERVAL ²
TIMESTAMP	GMTTIMESTAMP	-	INTERVAL ²
TIMESTAMP	INTERVAL	+, -	TIMESTAMP ³
GMTTIMESTAMP	TIMESTAMP	-	INTERVAL ²
GMTTIMESTAMP	GMTTIMESTAMP	-	INTERVAL ²
GMTTIMESTAMP	INTERVAL	+, -	GMTTIMESTAMP ³
INTERVAL	INTEGER	*, /	INTERVAL ⁴
INTERVAL	FLOAT	*, /	INTERVAL ⁴
INTERVAL	DECIMAL	*, /	INTERVAL ⁴
INTERVAL	TIME	+	TIME ³
INTERVAL	GMTTIME	+	GMTTIME ³
INTERVAL	DATE	+	DATE ³
INTERVAL	TIMESTAMP	+	TIMESTAMP ³
INTERVAL	GMTTIMESTAMP	+	GMTTIMESTAMP ³

Left operand data type	Right operand data type	Supported operators	Result data type
Notes:			
1. The operand that does not match the data type of the result is cast to the data type of the result before the operation proceeds. For example, if the left operand to an addition operator is an INTEGER, and the right operand is a FLOAT, the left operand is cast to a FLOAT before the addition operation is performed.			
2. Subtracting a (GMT)TIME value from a (GMT)TIME value, a DATE value from a DATE value, or a (GMT)TIMESTAMP value from a (GMT)TIMESTAMP value, results in an INTERVAL value representing the time interval between the two operands.			
3. Adding or subtracting an INTERVAL from a (GMT)TIME, DATE or (GMT)TIMESTAMP value results in a new value of the data type of the non-INTERVAL operand, representing the point in time represented by the original non-INTERVAL, plus or minus the length of time represented by the INTERVAL.			
4. Multiplying or dividing an INTERVAL by an INTEGER, FLOAT, or DECIMAL value results in a new INTERVAL representing the length of time represented by the original, multiplied or divided by the factor represented by the non-INTERVAL operand. For example, an INTERVAL value of 2 hours 16 minutes multiplied by a FLOAT value of 2.5 results in a new INTERVAL value of 5 hours 40 minutes. The intermediate calculations involved in multiplying or dividing the original INTERVAL are carried out in the data type of the non-INTERVAL, but the individual fields of the INTERVAL (such as HOUR, YEAR, and so on) are always integral, so some rounding errors might occur.			

Implicit CASTs for assignment

Values can be assigned to one of three entities.

A message field (or equivalent in an exception or destination list)

Support for implicit conversion between the WebSphere Message Broker data types and the message (in its bitstream form) depends on the appropriate parser. For example, the XML parser casts everything as character strings before inserting them into the WebSphere MQ message.

A field in a database table

WebSphere Message Broker converts each of its data types into a suitable standard SQL C data type, as detailed in the following table. Conversion between this standard SQL C data type, and the data types supported by each DBMS, depends on the DBMS. Consult your DBMS documentation for more details.

The following table lists the available conversions from WebSphere Message Broker to SQL data types

WebSphere Message Broker data type	SQL data type
NULL, or unknown or invalid value	SQL_NULL_DATA
BOOLEAN	SQL_C_BIT
INTEGER	SQL_C_LONG
FLOAT	SQL_C_DOUBLE
DECIMAL	SQL_C_CHAR ¹
CHARACTER	SQL_C_CHAR
TIME	SQL_C_TIME
GMTTIME	SQL_C_TIME
DATE	SQL_C_DATE
TIMESTAMP	SQL_C_TIMESTAMP
GMTTIMESTAMP	SQL_C_DATE

WebSphere Message Broker data type	SQL data type
INTERVAL	Not supported ²
BLOB	SQL_C_BINARY
BIT	Not supported ²
Notes:	
<ol style="list-style-type: none"> 1. For convenience, DECIMAL values are passed to the DBMS in character form. 2. There is no suitable standard SQL C data type for INTERVAL or BIT. Cast these to another data type, such as CHARACTER, if you need to assign them to a database field. 	

A scalar variable

When assigning to a scalar variable, if the data type of the value being assigned and that of the target variable data type are different, an implicit cast is attempted with the same restrictions and behavior as specified for the explicit CAST function. The only exception is when the data type of the variable is INTERVAL or DECIMAL.

In both these cases, the value being assigned is first cast to a CHARACTER value, and an attempt is made to cast the CHARACTER value to an INTERVAL or DECIMAL. This is because INTERVAL requires a qualifier and DECIMAL requires a precision and scale. These must be specified in the explicit cast, but must be obtained from the character string when implicitly casting. Therefore, a further restriction is that when implicitly casting to an INTERVAL variable, the character string must be of the form INTERVAL '<values>' <qualifier>. The shortened <values> form that is acceptable for the explicit cast is not acceptable here.

Data types of values from external sources

There are two external sources from which data can be extracted by ESQL:

- Message fields
- Database columns

The ESQL data type of message fields depends on the type of the message (XML for example), and the parser used to parse it. The ESQL data type of the value returned by a database column reference depends on the data type of the column in the database.

The following table shows which ESQL data types the various built-in DBMS data types are cast to, when they are accessed by WebSphere Message Broker.

The DBMS products are DB2 (version shipped with the product), SQL Server Version 7.0, Sybase Version 12.0, and Oracle Version 8.1.5

	DB2	SQL Server and Sybase	Oracle
BOOLEAN		BIT	
INTEGER	SMALLINT, INTEGER, BIGINT	INT, SMALLINT, TINYINT	
FLOAT	REAL, DOUBLE	FLOAT, REAL	NUMBER() ¹
DECIMAL	DECIMAL	DECIMAL, NUMERIC, MONEY, SMALLMONEY	NUMBER(P) ¹ , NUMBER(P,S) ¹

	DB2	SQL Server and Sybase	Oracle
CHARACTER	CHAR, VARCHAR, CLOB	CHAR, VARCHAR, TEXT	CHAR, NCHAR, VARCHAR2, NVARCHAR2, ROWID, UROWID, LONG, CLOB,
TIME	TIME		
GMTTIME			
DATE	DATE		
TIMESTAMP	TIMESTAMP	DATETIME, SMALLDATETIME	DATE
GMTTIMESTAMP			
INTERVAL			
BLOB	BLOB	BINARY, VARBINARY, TIMESTAMP, IMAGE, UNIQUEIDENTIFIER	RAW LONG, RAW BLOB
BIT			
Not supported	DATALINK, GRAPHIC, VARGRAPHIC, DBCLOB	NTEXT, NCHAR, NVARCHAR	NCLOB, BFILE

Note:

1. If an Oracle database column with NUMBER data type is defined with an explicit precision (P) and scale (S), it is cast to an ESQL DECIMAL value; otherwise it is cast to a FLOAT.

For example, an ESQL statement like this:

```
SET OutputRoot.xxx[]
= (SELECT T.department FROM Database.personnel AS T);
```

where Database.personnel resolves to a TINYINT column in an SQL Server database table, results in a list of ESQL INTEGER values being assigned to OutputRoot.xxx.

By contrast, an identical query, where Database.personnel resolves to a NUMBER() column in an Oracle database, results in a list of ESQL FLOAT values being assigned to OutputRoot.xxx.

Miscellaneous ESQL functions

This topic lists the miscellaneous ESQL functions and covers the following:

“COALESCE function”

“NULLIF function” on page 980

“PASSTHRU function” on page 980

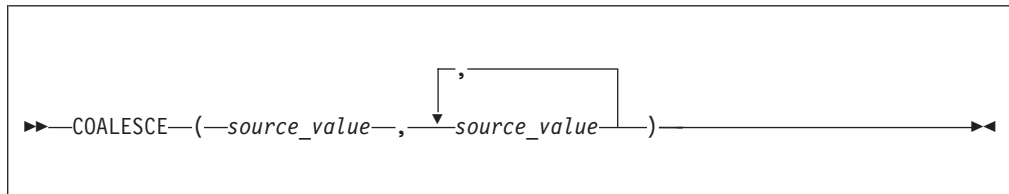
“UIDASBLOB function” on page 982

“UIDASCHAR function” on page 983

COALESCE function

COALESCE is a miscellaneous function that lets you provide default values for fields.

Syntax



The COALESCE function evaluates its parameters in order and returns the first one that is not NULL. The result is NULL if, and only if, all the arguments are NULL. The parameters can be of any scalar type, but they need not all be of the same type.

Use the COALESCE function to provide a default value for a field, which might not exist in a message. For example, the expression:

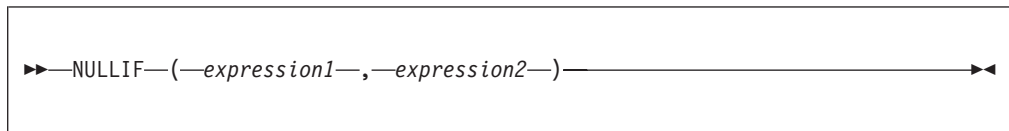
```
COALESCE(Body.Salary, 0)
```

returns the value of the Salary field in the message if it exists, or 0 (zero) if that field does not exist.

NULLIF function

NULLIF is a miscellaneous function that returns a NULL value if the arguments are equal.

Syntax



The NULLIF function returns a NULL value if the arguments are equal; otherwise, it returns the value of the first argument. The arguments must be comparable. The result of using NULLIF(e1,e2) is the same as using the expression:

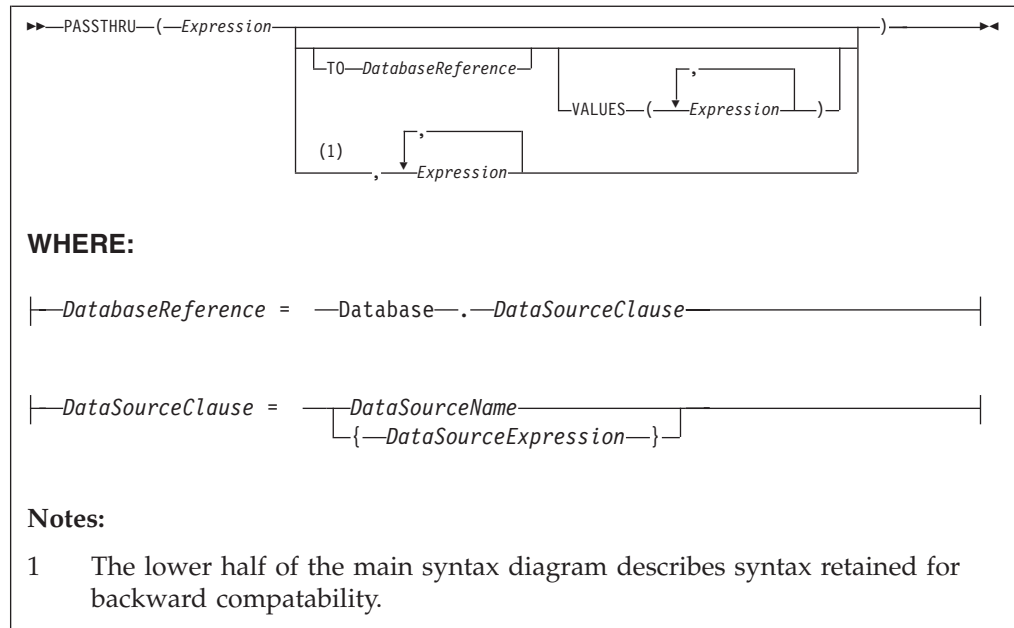
```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

When e1=e2 evaluates to unknown (because one or both of the arguments is NULL), NULLIF returns the value of the first argument.

PASSTHRU function

The PASSTHRU function evaluates an expression and executes the resulting character string as a database statement, returning a result set.

The PASSTHRU function is similar to the PASSTHRU statement, which is described in “PASSTHRU statement” on page 872.



Usage

The main use of the PASSTHRU function is to issue complex SELECTs, not currently supported by the broker, to databases. (Examples of complex SELECTs not currently supported by the broker are those containing GROUP BY or HAVING clauses.)

The first expression is evaluated and the resulting character string is passed to the database pointed to by *DatabaseReference* (in the TO clause) for execution. If the TO clause is not specified, the database pointed to by the node's data source attribute is used.

Use question marks (?) in the database string to denote parameters. The parameter values are supplied by the VALUES clause.

If the VALUES clause is specified, its expressions are evaluated and passed to the database as parameters; (that is, their values are substituted for the question marks in the database statement).

If there is only one VALUE expression, the result may or may not be a list. If it is a list, the list's scalar values are substituted for the question marks, sequentially. If it is not a list, the single scalar value is substituted for the (single) question mark in the database statement. If there is more than one VALUE expression, none of the expressions should evaluate to a list. Their scalar values are substituted for the question marks, sequentially.

Because the database statement is constructed by the user program, there is no absolute need to use parameter markers (that is, the question marks) or the VALUES clause, because the whole of the database statement could be supplied, as a literal string, by the program. However, it is recommended that you use parameter markers whenever possible, because this reduces the number of different statements that need to be prepared and stored in the database and the broker.

Database reference

A database reference is a special case of the field references used to refer to message trees. It consists of the word “Database” followed by a data source name (that is, the name of a database instance).

You can specify the data source name directly or by an expression enclosed in braces ({...}). A directly-specified data source name is subject to name substitution. That is, if the name used has been declared to be a known name, the value of the declared name is used rather than the name itself (see “DECLARE statement” on page 851).

Handling errors

It is possible for errors to occur during PASSTHRU operations. For example, the database may not be operational or the statement may be invalid. In these cases, an exception is thrown (unless the node has its throw exception on database error property set to FALSE). These exceptions set appropriate SQL code, state, native error, and error text values and can be dealt with by error handlers (see the DECLARE HANDLER statement).

For further information about handling database errors, see “Capturing database state” on page 219.

Example

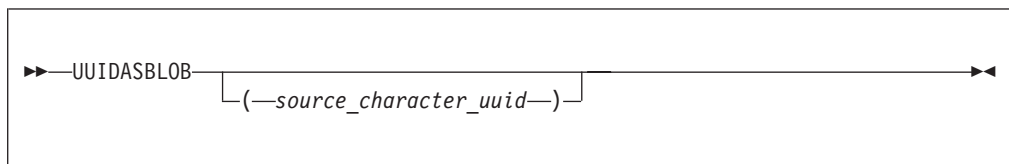
The following example performs a SELECT on table “Table1” in schema “Schema1” in database DSN1, passing two parameters to the WHERE clause and asking for the result set to be ordered in ascending name order. The result set is assigned to the SelectResult folder:

```
SET OutputRoot.XML.Data.SelectResult.Row[] =  
  PASSTHRU('SELECT R.* FROM Schema1.Table1 AS R WHERE R.Name = ? OR R.Name =  
    ? ORDER BY Name'  
  TO Database.DSN1  
  VALUES ('Name1', 'Name4'));
```

UIDASBLOB function

UIDASBLOB is a miscellaneous function that returns universally unique identifiers (UUIDs) as BLOBs.

Syntax



If (*source_character_uuid*) is not specified, UIDASBLOB creates a new UUID and returns it as a BLOB.

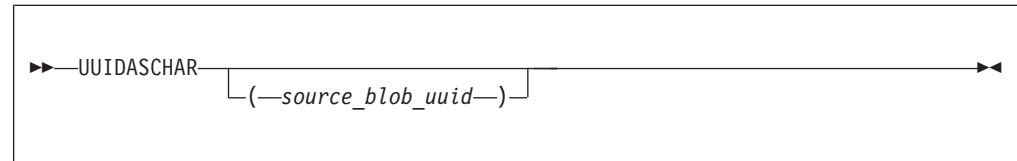
If (*source_character_uuid*) is specified, UIDASBLOB converts an existing character UUID in the form dddddd_dddd_dddd_ddddddddd to the BLOB form. An exception is thrown if the parameter is not of the expected form.

The result is NULL if a NULL parameter is supplied.

UUIDASCHAR function

UUIDASCHAR is a miscellaneous function that returns universally unique identifiers (UUIDs) as CHARACTER values.

Syntax



If (*source_character_uuid*) is not specified, UUIDASCHAR creates a new UUID and returns it as a CHARACTER value.

If (*source_character_uuid*) is specified, UUIDASCHAR converts an existing BLOB UUID to the character form.

The result is NULL if a NULL parameter is supplied.

Broker properties accessible from ESQL and Java

For an overview of broker properties, see “Broker properties” on page 49.

The following table shows the broker, flow, and node properties that are accessible from ESQL. The table’s fourth column indicates whether the properties are also accessible from Java nodes.

If a property is listed as being accessible from Java nodes (fourth column), it is accessible from Java nodes *only*, *not* from Java routines called as ESQL functions or procedures.

Property type	Property name	Return type	From Java nodes?	What is it?
General broker properties ⁴	BrokerDataSourceUserId	Character	Yes. ¹	The data source user ID used by the broker.
	BrokerDataSource	Character	No.	The ODBC Data Source Name (DSN) of the database that contains the broker's tables.
	BrokerName	Character	Yes. ²	The name of the broker.
	BrokerUserId	Character	No	The user ID that the broker uses to access its database tables.
	BrokerVersion	Character	No	The 4-character version number of the broker (see "BrokerVersion" on page 986 below).
	ExecutionGroupLabel	Character	Yes. ³	The label of the Execution Group (a human-readable name).
	ExecutionGroupName	Character	No	The name of the Execution Group (often a UUID identifier).
	Family	Character	No	The generic name of the software platform that the broker is running on ('WINDOWS', 'UNIX', or 'ZOS').
	ProcessId	Integer	No	The process identifier (PID) of the DataFlowEngine.
	QueueManagerName	Character	Yes. ⁵	The name of the MQ queue manager to which the broker is connected.
WorkPath	Character	No.	The (optional) directory in which working files for this broker are stored.	
Flow properties	AdditionalInstances	Integer	No	The number of additional threads that the broker can use to service the message flow.
	CommitCount	Integer	No	How many input messages are processed by the message flow before a syncpoint is taken.
	CommitInterval	Integer	No	The time interval at which a commit is taken when the <i>CommitCount</i> property is greater than 1 (that is, where the message flow is batching messages), but the number of messages processed has not reached the value of the <i>CommitCount</i> property.
	CoordinatedTransaction	Boolean	Yes. ⁶	Whether or not the message flow is processed as a global transaction, coordinated by WebSphere MQ.
	MessageFlowLabel	Character	Yes. ⁷	The name of the flow.

Property type	Property name	Return type	From Java nodes?	What is it?
Node properties	DataSource	Character	No	The ODBC Data Source Name (DSN) of the database in which the user tables are created.
	DataSourceUserId	Character	No	The user ID that the broker uses to access the database user tables.
	MessageOptions	Integer (64-bit)	No	The bitstream and validation options in force.
	NodeLabel	Character	Yes. ⁸	The name of the node.
	NodeType	Character	No	The type of node (Compute, Filter, or Database).
	ThrowExceptionOnDatabaseError	Boolean	No	Whether the broker generates an exception when a database error is detected.
	Transaction	Character	No	The type of transaction (Automatic or commit) used to access a database from this node.
	TreatWarningsAsErrors	Boolean	No	Whether database warning messages are treated as errors and cause the output message to be propagated to the failure terminal.

Notes:

1. Accessible through:
 - a. `MbNode.getBroker()`
 - b. `MbBroker.getDataSourceUserId()`
2. Accessible through:
 - a. `MbNode.getBroker()`
 - b. `MbBroker.getName()`
3. Accessible through:
 - a. `MbNode.getExecutionGroup()`
 - b. `MbExecutionGroup.getName()`
4. The only broker-defined properties that can be used in a Trace node are those in the “General broker properties” group. For example, you could specify the Pattern setting of a Trace node as:

```
#### Start Trace Input Message
Time: ${CURRENT_TIMESTAMP}
Broker: ${BrokerName} Version: ${BrokerVersion} Platform: ${Family}
ProcessID: ${ProcessId} BrokerUserId: ${BrokerUserId}
ExecutionGroupLabel: ${ExecutionGroupLabel}
Transaction: ${Transaction}
Root Tree: ${Root}
#### End Trace Input Message
```
5. Accessible through:
 - a. `MbNode.getBroker()`
 - b. `MbBroker.getQueueManagerName()`
6. Accessible through:
 - a. `MbNode.getMessageFlow()`
 - b. `MbMessageFlow.isCoordinatedTransaction()`
7. Accessible through:
 - a. `MbNode.getMessageFlow()`
 - b. `MbMessageFlow.getName()`

8. Accessible through `MbNode.getName()`

BrokerVersion

The `BrokerVersion` property contains a 4-character code that indicates the version of the broker. The code is based on the IBM Version/Release/Modification/Fix pack (VRMF) product-numbering system. The VRMF code works like this:

- V** The Version number. A Version is a separate IBM licensed program that usually has significant new code or new function. Each version has its own license, terms, and conditions.
- R** The Release number. A Release is a distribution of new function and authorized program analysis report (APAR) fixes for an existing product.
- M** The Modification number. A Modification is new function added to an existing product, and is delivered separately from an announced Version or Release.
- F** The Fix pack number. Fix packs contain defect and APAR fixes. They do not contain new function.

A fix pack is cumulative: that is, it contains all the fixes shipped in previous maintenance to the release, including previous fix packs. It can be applied on top of any previously-shipped maintenance to bring the system up to the current fix pack level.

Special characters, case sensitivity, and comments in ESQL

This topic describes the special characters used in ESQL, case sensitivity, and how comments are handled in the following sections:

- “Special characters”
- “Case sensitivity of ESQL syntax” on page 987
- “Comments” on page 987

Special characters

Symbol	Name	Usage
;	semicolon	End of ESQL statement
.	period	Field reference separator or decimal point
=	equals	Comparison or assignment
>	greater than	Comparison
<	less than	Comparison
[]	square brackets	Array subscript
'	single quotation mark	Delimit string, date-time, and decimal literals Note, that to escape a single quotation mark inside a string literal, you must use two single quotation marks.
	double vertical bar	Concatenation
()	parentheses	Expression delimiter

Symbol	Name	Usage
"	quotation mark	Identifier delimiter
*	asterisk	Any name or multiply
+	plus	Arithmetic add
-	minus	Arithmetic subtract, date separator, or negation
/	forward slash	Arithmetic divide
_	underscore	LIKE single wild card
%	percent	LIKE multiple wild card
\	backslash	LIKE escape character
:	colon	Name space and Time literal separator
,	comma	List separator
<>	less than greater than	Not equals
--	double minus	ESQL single line comment
/* */	slash asterisk asterisk slash	ESQL multiline comment
?	question mark	Substitution variable in PASSTHRU
<=	less than or equal	Comparison
>=	greater than or equal	Comparison
/!{ }!*/	executable comment	Bypass tools check

Case sensitivity of ESQL syntax

The case of ESQL statements is:

- Case sensitive in field reference literals
- Not case sensitive in ESQL language words

Comments

ESQL has two types of comment: single line and multiple line. A single line comment starts with the characters -- and ends at the end of the line.

In arithmetic expressions you must take care not to initiate a line comment accidentally. For example, consider the expression:

```
1 - -2
```

Removing all white space from the expression results in:

```
1--2
```

which is interpreted as the number 1, followed by a line comment.

A multiple line comment starts with /* anywhere in ESQL and ends with */.

ESQL reserved keywords

The following keywords are reserved in uppercase, lowercase, or mixed case. You cannot use these keywords for variable names. However, you *can* use reserved keywords as names in a field reference.

ALL	ASYMMETRIC	BOTH
CASE	DISTINCT	FROM
ITEM	LEADING	NOT
SYMMETRIC	TRAILING	WHEN

ESQL non-reserved keywords

The following keywords are used in the ESQL language but are not reserved. We recommend that you do not use them for variable, function, or procedure names (in any combination of upper and lower case) because, if you do, the code can become difficult to understand.

- AND
- ANY
- AS
- ATOMIC
- ATTACH
- BEGIN
- BETWEEN
- BIT
- BLOB
- BOOLEAN
- BY
- CALL
- CATALOG
- CCSID
- CHAR
- CHARACTER
- COMMIT
- COMPUTE
- CONDITION
- CONSTANT
- CONTINUE
- COORDINATED
- COUNT
- CREATE
- CURRENT_DATE
- CURRENT_GMTDATE
- CURRENT_GMTTIME
- CURRENT_GMTTIMESTAMP
- CURRENT_TIME
- CURRENT_TIMESTAMP
- DATA
- DATABASE
- DATE
- DAY
- DAYOFWEEK
- DAYOFYEAR

- DAYS
- DECIMAL
- DECLARE
- DEFAULT
- DELETE
- DETACH
- DO
- DOMAIN
- DYNAMIC
- ELSE
- ELSEIF
- ENCODING
- END
- ENVIRONMENT
- ESCAPE
- ESQL
- EVAL
- EVENT
- EXCEPTION
- EXISTS
- EXIT
- EXTERNAL
- FALSE
- FIELD
- FILTER
- FINALIZE
- FIRSTCHILD
- FLOAT
- FOR
- FORMAT
- FOUND
- FULL
- FUNCTION
- GMTTIME
- GMTTIMESTAMP
- GROUP
- HANDLER
- HAVING
- HOUR
- IDENTITY
- IF
- IN
- INF
- INFINITY
- INOUT
- INSERT
- INT
- INTEGER
- INTERVAL
- INTO
- IS
- ISLEAPYEAR
- ITERATE
- JAVA
- LABEL
- LANGUAGE

- LAST
- LASTCHILD
- LEAVE
- LIKE
- LIST
- LOCALTIMEZONE
- LOG
- LOOP
- MAX
- MESSAGE
- MIN
- MINUTE
- MODIFIES
- MODULE
- MONTH
- MONTHS
- MOVE
- NAME
- NAMESPACE
- NAN
- NEXTSIBLING
- NONE
- NULL
- NUM
- NUMBER
- OF
- OPTIONS
- OR
- ORDER
- OUT
- PARSE
- PASSTHRU
- PATH
- PLACING
- PREVIOUSIBLING
- PROCEDURE
- PROPAGATE
- QUARTEROFYEAR
- QUARTERS
- READS
- REFERENCE
- REPEAT
- RESIGNAL
- RESULT
- RETURN
- RETURNS
- ROLLBACK
- ROW
- SAMEFIELD
- SCHEMA
- SECOND
- SELECT
- SET
- SETS
- SEVERITY
- SHARED

- SHORT
- SOME
- SQL
- SQLCODE
- SQLERRORTXT
- SQLEXCEPTION
- SQLNATIVEERROR
- SQLSTATE
- SQLWARNING
- SUM
- TERMINAL
- THE
- THEN
- THROW
- TIME
- TIMESTAMP
- TO
- TRACE
- TRUE
- TYPE
- UNCOORDINATED
- UNKNOWN
- UNTIL
- UPDATE
- USER
- UUIDASBLOB
- UUIDASCHAR
- VALUE
- VALUES
- WEEKOFMONTH
- WEEKOFYEAR
- WEEKS
- WHERE
- WHILE
- YEAR

Example message

This topic defines the example message that is used in many of the examples throughout the information center.

The example message is:

```

<Invoice>
<InvoiceNo>300524</InvoiceNo>
<InvoiceDate>2000-12-07</InvoiceDate>
<InvoiceTime>12:40:00</InvoiceTime>
<TillNumber>3</TillNumber>
<Cashier StaffNo="089">Mary</Cashier>
<Customer>
  <FirstName>Andrew</FirstName>
  <LastName>Smith</LastName>
  <Title>Mr</Title>
  <DOB>20-01-70</DOB>
  <PhoneHome>01962818000</PhoneHome>
  <PhoneWork/>
  <Billing>
    <Address>14 High Street</Address>
    <Address>Hursley Village</Address>
  
```

```

        <Address>Hampshire</Address>
        <PostCode>S0213JR</PostCode>
    </Billing>
</Customer>
<Payment>
    <CardType>Visa</CardType>
    <CardNo>4921682832258418</CardNo>
    <CardName>Mr Andrew J. Smith</CardName>
    <Valid>1200</Valid>
    <Expires>1101</Expires>
</Payment>
<Purchases>
    <Item>
        <Title Category="Computer" Form="Paperback" Edition="2">The XML Companion
    </Title>
        <ISBN>0201674866</ISBN>
        <Author>Neil Bradley</Author>
        <Publisher>Addison-Wesley</Publisher>
        <PublishDate>October 1999</PublishDate>
        <UnitPrice>27.95</UnitPrice>
        <Quantity>2</Quantity>
    </Item>
    <Item>
        <Title Category="Computer" Form="Paperback" Edition="2">A Complete Guide
to DB2 Universal Database</Title>
        <ISBN>1558604820</ISBN>
        <Author>Don Chamberlin</Author>
        <Publisher>Morgan Kaufmann Publishers</Publisher>
        <PublishDate>April 1998</PublishDate>
        <UnitPrice>42.95</UnitPrice>
        <Quantity>1</Quantity>
    </Item>
    <Item>
        <Title Category="Computer" Form="Hardcover" Edition="0">JAVA 2 Developers
Handbook</Title>
        <ISBN>0782121799</ISBN>
        <Author>Philip Heller, Simon Roberts </Author>
        <Publisher>Sybex, Inc.</Publisher>
        <PublishDate>September 1998</PublishDate>
        <UnitPrice>59.99</UnitPrice>
        <Quantity>1</Quantity>
    </Item>
</Purchases>
<StoreRecords/>
<DirectMail/>
<Error/>
</Invoice>

```

For a diagrammatic representation of this message, and for examples of how this message can be manipulated with ESQL statements and functions, refer to “Writing ESQL” on page 168.

Flow application debugger

This section includes the following topics:

- “Java Debugger”
- “Flow debugger shortcuts”
- “Flow debugger icons and symbols” on page 994

Java Debugger

For information about the Java debugger, refer to the Java Development User Guide plug-in - Debugger.

Flow debugger shortcuts

This topic describes the shortcut keys that you can use in the flow debugger views and windows.

The keys are shown as a pair that you press together, followed by a subsequent key, for example Shift-F10, C means hold the Shift key down and press F10, then release both and press key C.

When you press Shift-F10 and release, the contextual pop-up menu appears, listing the shortcut keys that are available.

This topic describes the following subjects:

- “Debug view”
- “Breakpoints view” on page 994
- “Flow Breakpoint Properties dialog” on page 994
- “Variables view” on page 994

Debug view

Key combination	Function
Shift-F10, A	Attach to the selected flow engine
Shift-F10, D	Detach from the selected flow engine
Shift-F10, R	Refresh the list of flows for the selected flow engine
Shift-F10, R	Resume the flow execution
Shift-F10, C	Run the flow to completion
Shift-F10, I	Step into
Shift-F10, U	Step out of
Shift-F10, O	Step over
Shift-F10, N	Step into source code

Breakpoints view

Key combination	Function
Shift-F10, E	Enable the selected breakpoints
Shift-F10, D	Disable the selected breakpoints
Shift-F10, O	Remove the selected breakpoints
Shift-F10, L	Remove all breakpoints
Shift-F10, P	Open the Properties dialog

Flow Breakpoint Properties dialog

Key combination	Function
E	Enable the breakpoint
Alt-R, <space>	Restrict the breakpoint to the selected flow instances

Variables view

Key combination	Function
Shift-F10, C	Change the value of the selected flow variable
Shift-F10, S	Show the type names of all flow variables
Shift-F10, H	Hide the type names of all flow variables



Flow debugger icons and symbols

This topic describes the icons and symbols used in the Debug perspective and its views:















- “Debug perspective”
- “Debug view” on page 995
- “Message Flow editor” on page 995
- “Breakpoints view” on page 996
- “Variables view” on page 996

Debug perspective

These icons and symbols are used in the Debug perspective outside any individual view.






Icon or Symbol	Description
	Debug perspective (symbol)
	Attach to Flow runtime (icon)

Debug view






Icon or Symbol	Description
	Debug view (symbol)
	Flow engine (symbol)
	Flow (symbol)
	Flow instance paused (symbol)
	Flow instance running (symbol)
	Flow instance terminated (symbol)
	Stack frame (symbol)
	Detach from the selected flow engine (icon)
	Resume flow execution (icon)
	Run the flow to completion (icon)
	Step into subflow (icon)
	Step over node (icon)
	Step out of subflow (icon)
	Step into source code (icon)

Message Flow editor

These icons and symbols in the message flow editor are specific to the flow debugger.









Icon or Symbol	Description
	Enabled breakpoint (symbol)
	Disabled breakpoint (symbol)
	Paused at breakpoint (symbol)
	Source code available (symbol)
	Error or exception (symbol)

Breakpoints view

Icon or Symbol	Description
	Breakpoints view (symbol)
	Enabled breakpoint (symbol)
	Disabled breakpoint (symbol)
	Remove selected breakpoints (icon)
	Remove all breakpoints (icon)

Variables view

These icons and symbols in the Variables view are specific to ESQL.

Icon or Symbol	Description
	Variable view (symbol)
	Tree reference variable (symbol)
	Message (symbol)
	ESQL reference variable (symbol)
	ESQL constant (symbol)
	ESQL scalar variable (symbol)
	ESQL schema variable (symbol)
	ESQL module variable (symbol)

Part 5. Appendixes

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.*

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032,
Japan*

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information includes examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not

been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) *(your company name)* *(year)*. Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX	CICS	Cloudscape
DB2	DB2 Connect	DB2 Universal Database
developerWorks	Domino	
Everyplace	FFST	First Failure Support Technology
IBM	IBMLink	IMS
IMS/ESA	iSeries	Language Environment
Lotus	MQSeries	MVS
NetView	OS/400	OS/390
pSeries	RACF	Rational
Redbooks	RETAIN	RS/6000
SupportPac	Tivoli	VisualAge
WebSphere	xSeries	z/OS
zSeries		

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

A

- accessing headers
 - Java 297
- accounting and statistics data 54
 - accounting origin 56
 - collecting 374
 - collection options 55
 - output formats 57
 - parameters, modifying 379
 - parameters, viewing 378
 - resetting archive data 380
 - setting accounting origin 376
 - starting 374
 - stopping 377
- accounting origin 56
 - setting 376
- AggregateControl node 483
- AggregateReply node 485
- AggregateRequest node 488
- application clients
 - request-response
 - MQGet node 407
- archive data 55
 - resetting 380

B

- bar file 424
- bar files
 - creating 431
 - deploying 434
 - editing
 - manually 433
 - properties 433
 - message flows, adding 432
 - message sets, adding 432
- bend points, message flows 8
- breakpoints 462
 - adding 462
 - disabling 464
 - enabling 464
 - removing 465
 - restricting 463
- broker
 - cancel deployment 428
- broker archive 424
 - configurable properties 425
 - deployment 422
- broker archive files
 - creating 431
 - deploying 434
 - editing
 - manually 433
 - properties 433
 - message flows, adding 432
 - message sets, adding 432
- broker configuration deployment 426
- broker configuration, deploying 436
- broker properties, message flow 49, 151
- broker schema 52

- brokers
 - deployed flows, querying 473

C

- cancel deployment 428
- Check node 490
- code dependencies
 - Java 287
- code pages
 - conversion 92
 - converting with ESQL 198
- complete broker archive deployment 422
- complete topics deployment 427
- complete topology deployment 426
- Compute node 492
- conditional mappings, configuring 314
- conditional mappings, creating 322
- configurable properties
 - broker archive 425
- configurable properties, message flow 715
- connections
 - databases 700
 - listing 701
 - WebSphere MQ 699
 - connections, message flows 8
- copying headers
 - Java 298
- correlation names
 - logical message tree 23
 - XML constructs 762

D

- data conversion 92
- data source
 - z/OS
 - Compute node 778
 - Database node 778
- data types
 - BLOB message 739
 - DestinationData subtree 736
 - elements 735
 - fields 735
 - MRM message 738
 - Properties subtree 736
 - WebSphere MQ header fields 735
- database connections
 - listing 701
 - ODBC 700
 - quiescing 701
 - user database 700
- Database node 500
- databases
 - code page support 702
 - configuring coordinated message flows 96
 - connections 700

- databases (*continued*)
 - DBCS restrictions 702
 - deadlock 93
 - Java 301
 - listing connections 701
 - quiescing 701
 - stored procedures in ESQL 212
- DataDelete node 504
- DataInsert node 507
- DataUpdate node 510
- DBCS
 - database restrictions 702
- debugger 452
 - icons and symbols 994
 - keyboard shortcuts 993
 - starting 455
- debugging 452
 - data 470
 - ESQL 471
 - Java 472
 - mappings 472
 - messages 470
 - dequeuing 461
 - enqueueing 459
 - icons and symbols 994
 - keyboard shortcuts 993
 - message flows 451
 - stepping through 466
 - delta topics deployment 427
 - delta topology deployment 426
 - deployment 419
 - broker archive (bar) files 434
 - broker configuration 436
 - canceling 443
 - checking results 441
 - message flow application 429
 - overview 419
 - broker archive (bar) files 424
 - broker configuration 426
 - cancel 428
 - configurable properties 425
 - message flow application 422
 - topics 427
 - topology 426
 - publish/subscribe topics
 - hierarchy 439
 - publish/subscribe topology 438
 - Version 5 or Version 6 authored ESQL to a Version 2.1 broker 158
 - dequeuing, using in debugging 461
 - Destination (LocalEnvironment), populating 194
 - developing
 - Java 285
 - displaying
 - version and keywords 131
 - domain
 - cancel deployment 428

E

editors

- Message Mapping 748

element definitions for message

- parsers 735

encoding 92

- enqueueing, using in debugging 459

Environment tree 17

- accessing with ESQL 195

errors

- Timeout Notification node 119

ESQL

- accessible from Java 49, 151

- accessing databases 86

- BLOB messages 277

- Broker attributes 49, 151

- code generation level 166

- converting EBCDIC NL to ASCII
CRLF 200

data

- casting 197

- converting 198

- transforming 197

- data types 147

database columns

- referencing 205

- selecting data from 207

- database content, changing 210

- database state, capturing 219

- database updates, committing 212

- databases, interacting with 204

- datetime representation 784

- debugging 471

- deploying Version 5 or Version 6 to a
Version 2.1 broker 158

- Destination, populating 194

- developing 145

elements

- accessing 172

- setting or querying null 172

elements, multiple occurrences

- accessing known 176

- accessing unknown 177

- Environment tree, accessing 195

- errors, handling 214

- example message 991

- exception, throwing 218

- ExceptionList tree, accessing 196

- explicit null handling 172

- field references 151

- anonymous 178

- creating 179

- syntax 792

- field types, referencing 171

fields

- copying those that repeat 186

- creating new 180

- manipulating those that repeat in a
message tree 188

files

- copying 164

- creating 156

- deleting 168

- moving 165

- opening 158

- renaming 164

- saving 162

ESQL (continued)

- functions 153

- headers, accessing 189

- IDoc messages 275

- implicit null handling 172

- JMS messages 275

- keywords 204

- non-reserved 988

- reserved 988

- like-parser-copy 202

- LocalEnvironment tree, accessing 192

- message body data,

- manipulating 171

- message format, changing 202

- message tree parts, manipulating 189

- MIME messages 275

- modules 155

- MQMD header, accessing 189

- MQRFH2 header, accessing 190

- MRM domain messages

- handling large 235

- working with 233

- MRM domain messages, accessing

- attributes 223

- base types 232

- elements 221

- elements in groups 224

- embedded messages 227

- embedded simple types 230

- migrated objects 230

- mixed content 226

- multiple occurrences 222

- namespace-enabled messages 228

- MRM domain messages, null values

- querying 229

- setting 229

- multiple database tables,

- accessing 209

- nested statements 153

node

- creating 158

- deleting 167

- modifying 161

- numeric operators with datetime 183

- operators 152

- complex comparison 799

- logical 802

- numeric 803

- rules for operator precedence 804

- simple comparison 798

- string 804

- output messages, generating 182

- preferences, changing 166

- procedures 154

- Properties tree, accessing 190

- returns to SELECT, checking 211

- settings

- editor 166

- validation 167

- special characters 986

- statements 152

- stored procedures, invoking 212

- subfield, selecting 185

- syntax preference 780

- tailoring for different nodes 170

- time interval, calculating 184

- unlike-parser-copy 202

ESQL (continued)

- variables 147

XML messages

- attributes, accessing 239

- bit streams 266

- complex message,

- transforming 252

- data, translating 260

- DTD, accessing 242

- fields, ordering 247

- message and table data,

- joining 262

- message data, joining 261

- messages, constructing 248

- messages, handling large 255

- paths and types,

- manipulating 246

- scalar value, returning 258

- simple message, transforming 249

- XMLDecl, accessing 241

- XMLNS messages 267

- XMLNSC parser, manipulating

- messages using 269

ESQL data types

- BOOLEAN 780

database

- ROW 781

- Datetime 780

- DATE 781

- GMTTIME 782

- GMTTIMESTAMP 782

- INTERVAL 783

- TIME 782

- TIMESTAMP 782

- ESQL to Java, mapping of 790

- list of 780

- NULL 786

- numeric 786

- DECIMAL 787

- FLOAT 788

- INTEGER 788

- REFERENCE 789

- string 789

- BIT 789

- BLOB 789

- CHARACTER 790

- ESQL functions 889

CAST

- formatting and parsing dates as
strings 948

- formatting and parsing numbers as
strings 945

- formatting and parsing times as
strings 948

- complex 938

- CASE 939

- CAST 940

- Data types from external

- sources 978

- LIST constructor 961

- ROW and LIST combined 963

- ROW and LIST comparisons 963

- ROW constructor 960

- SELECT 954

- Supported casts 965

- database state 892

- SQLCODE 892

ESQL functions (*continued*)

- database state (*continued*)
 - SQLERRORTEXT 893
 - SQLNATIVEERROR 894
 - SQLSTATE 894
- datetime 897
 - CURRENT_DATE 900
 - CURRENT_GMTDATE 901
 - CURRENT_GMTTIME 901
 - CURRENT_GMTTIMESTAMP 901
 - CURRENT_TIME 900
 - CURRENT_TIMESTAMP 900
 - EXTRACT 898
 - LOCAL_TIMEZONE 902
- field 925
 - ASBITSTREAM 925
 - BITSTREAM 929
 - FIELDNAME 929
 - FIELDNAMESPACE 930
 - FIELDTYPE 930
 - FIELDVALUE 933
 - FOR 933
 - LASTMOVE 935
 - SAMEFIELD 935
- implicit casts 973
 - arithmetic operations 976
 - assignment 977
 - comparisons 973
- list 936
 - CARDINALITY 936
 - EXISTS 937
 - SINGULAR 937
 - THE 938
- miscellaneous 979
 - COALESCE 979
 - NULLIF 980
 - PASSTHRU 980
 - UIDASBLOB 982
 - UIDASCHAR 983
- numeric 902
 - ABS and ABSVAL 903
 - ACOS 904
 - ASIN 904
 - ATAN 904
 - ATAN2 904
 - BITAND 905
 - BITNOT 905
 - BITOR 906
 - BITXOR 906
 - CEIL and CEILING 907
 - COS 907
 - COSH 908
 - COT 908
 - DEGREES 908
 - EXP 908
 - FLOOR 909
 - LN and LOG 909
 - LOG10 910
 - MOD 910
 - POWER 911
 - RADIANS 911
 - RAND 911
 - ROUND 912
 - SIGN 912
 - SIN 913
 - SINH 913
 - SQRT 913

ESQL functions (*continued*)

- numeric (*continued*)
 - TAN 914
 - TANH 914
 - TRUNCATE 915
- string manipulation 915
 - LEFT 916
 - LENGTH 916
 - LOWER and LCASE 917
 - LTRIM 917
 - OVERLAY 918
 - POSITION 919
 - REPLACE 920
 - REPLICATE 920
 - RIGHT 921
 - RTRIM 921
 - SPACE 922
 - SUBSTRING 922
 - TRANSLATE 923
 - TRIM 924
 - UPPER and UCASE 925
- ESQL statements 804
 - ATTACH 806
 - BEGIN ... END 807
 - BROKER SCHEMA 810
 - PATH clause 811
 - CALL 813
 - CASE 816
 - CREATE 818
 - CREATE FUNCTION 826
 - CREATE MODULE 835
 - CREATE PROCEDURE 837
 - DECLARE 851
 - DECLARE HANDLER 856
 - DELETE 860
 - DELETE FROM 857
 - DETACH 860
 - EVAL 861
 - FOR 862
 - IF 863
 - INSERT 864
 - ITERATE 866
 - LEAVE 866
 - list of available 804
 - Local error handler 856
 - LOG 867
 - LOOP 869
 - MOVE 870
 - PASSTHRU 872
 - PROPAGATE 874
 - REPEAT 877
 - RESIGNAL 878
 - RETURN 878
 - SET 880
 - list type elements, working with 882
 - THROW 884
 - UPDATE 885
 - WHILE 888
- ESQL, overview of 779
- exception handling
 - Java 302
- ExceptionList tree 21
 - accessing with ESQL 196
- exceptions
 - message tree content 12

execution groups

- message flows, removing 445

 Extract node 513

F

field names, IDoc parser 740
 Filter node 515
 flow debugger 452

- ESQL nodes 454
- icons and symbols 994
- keyboard shortcuts 993
- mapping nodes 455
- user-defined source (Java) 455

 flow engine

- attaching to 456
- detaching from 474

 flow instances

- managing 473
- stepping through 466
 - resuming execution 466
 - running execution to completion 466
 - stepping into source code 469
 - stepping into subflows 467
 - stepping out of source code 469
 - stepping out of subflows 467
 - stepping over nodes 467
 - stepping over source code 468
- terminating 474

 FlowOrder node 519

G

global environment

- Java 299

H

headers 322
 HTTPInput node 521
 HTTPReply node 527
 HTTPRequest node 529

I

icons for flow debugger 994
 IDoc

- parser 34

 incremental broker archive

- deployment 422

 Input node 540

J

Java

- accessing attributes 299
- accessing elements 289
- accessing headers 297
- accessing the global environment 299
- code dependencies 287
- copying a message 292
- copying headers 298
- creating a filter 294
- creating a new message 292

- Java (*continued*)
 - creating code 286
 - creating elements 293
 - deploying code 288
 - developing 285
 - exception handling 302
 - interacting with databases 301
 - keywords 300
 - logging errors 302
 - managing files 286
 - manipulating messages 289
 - MQMD 298
 - MQRFH2 298
 - opening files 287
 - propagating a message 295
 - saving files 287
 - setting elements 292
 - transforming messages 291
 - updating the Local Environment 299
 - user-defined properties 300
 - using XPath 295
 - writing 288
- Java debugger 993
- Java, broker attributes accessible from 49, 151
- Java, debugging 472
- JavaCompute node 542
- JMSInput node 546
- JMSMQTransform node 557
- JMSOutput node 558

K

- keyboard shortcuts
 - flow debugger 993
- keyword
 - displaying 131
- keywords 480
 - description properties 479
 - ESQL 204
 - Java 300
 - subflows 72
 - XSL stylesheet 670

L

- Label node 565
- list of available 889
- local environment
 - Java 299
- LocalEnvironment tree 18
 - accessing with ESQL 192
 - populating Destination 194
 - using as scratchpad 193
- logical message tree
 - contents after exception 12
 - correlation names 23
 - Environment tree 17
 - ExceptionList tree 21
 - LocalEnvironment tree 18
 - message body 15
 - Properties folder 15
 - structure 14

M

- manipulating messages
 - Java 289
- map file, creating 306
- Mapping node 567
 - casts 758
 - functions 758
 - syntax 757
- mappings 322
 - by name 309
 - by selection 309
 - conditional
 - configuring 314
 - creating 322
 - configuring 307
 - creating 306
 - database
 - adding 309
 - source 307
 - deleting source and target 313
 - derived types 304
 - developing 302
 - examples 326
 - from database tables 311
 - from source messages 310
 - list types 304
 - LocalEnvironment, configuring 316
 - map file, creating 306
 - Mapping node casts 758
 - Mapping node functions 758
 - Mapping node syntax 757
 - Message Mapping editor 748
 - Edit pane 754
 - Source pane 749
 - Spreadsheet pane 755
 - Target pane 752
 - message, adding 308
 - migrating 759
 - restrictions 760
 - overview 303
 - populate 307
 - repeating elements, configuring 315
 - restrictions 324
 - scenarios 326
 - schema structure 304
 - SOAP 321
 - statements, order 322
 - submaps 316
 - calling 318
 - creating 316
 - wildcard source 317
 - subroutines 316
 - calling 319, 320
 - from ESQL 319
 - user-defined, calling 320
 - substitution groups 304
 - target, setting the value
 - to a constant 312
 - using a function 313
 - using an expression 313
 - union types 304
 - wildcards 304
 - mappings, debugging 472
 - message body 15
 - ESQL, accessing with 171
 - message flow
 - stylesheet keywords 670

- message flow application
 - deployment 422
- message flow application, deploying 429
- message flow nodes 482
 - AggregateControl 483
 - AggregateReply 485
 - AggregateRequest 488
 - Check 490
 - Compute 492
 - Database 500
 - DataDelete 504
 - DataInsert 507
 - DataUpdate 510
 - Extract 513
 - Filter 515
 - FlowOrder 519
 - HTTPInput 521
 - HTTPReply 527
 - HTTPRequest 529
 - Input 540
 - JavaCompute 542
 - JMSInput 546
 - JMSMQTransform 557
 - JMSOutput 558
 - Label 565
 - Mapping 567
 - MQeInput 573
 - MQeOutput 580
 - MQGet 584
 - MQInput 593
 - MQJMSTransform 602
 - MQOptimizedFlow 603
 - MQOutput 605
 - MQReply 612
 - Output 615
 - Passthrough 617
 - Publication 619
 - Real-timeInput 621
 - Real-timeOptimizedFlow 623
 - ResetContentDescriptor 626
 - RouteToLabel 630
 - SCADAInput 632
 - SCADAOutput 639
 - Throw 642
 - Timeout Control 644
 - Timeout Notification 647
 - Trace 652
 - TryCatch 656
 - Validate 658
 - Warehouse 661
 - XMLTransformation 665
- message flows 3
 - accessing databases 84, 86, 88
 - accounting and statistics data 54
 - accounting origin 56
 - collecting 374
 - collection options 55
 - details 717
 - example output 730
 - output data formats 718
 - output formats 57
 - parameters, modifying 379
 - parameters, viewing 378
 - resetting archive data 380
 - setting accounting origin 376
 - starting 374
 - stopping 377

- message flows (*continued*)
 - aggregation 90
 - database deadlocks, resolving 400
 - exceptions, handling 400, 402
 - fan-in flow, creating 389
 - fan-out and fan-in flows, associating 393
 - fan-out flow, creating 385
 - multiple AggregateControl nodes 395
 - requests and responses, correlating 396
 - timeouts, setting 394
 - unknown and timeout message exceptions 402
 - bend points 8
 - adding 143
 - removing 143
 - broker archive (bar) file, adding to 432
 - broker properties 49, 151
 - broker schemas
 - creating 125
 - deleting 131
 - built-in nodes 482
 - Chinese code page GB18030 699
 - cluster queues 77
 - code page support 671
 - configurable properties 715
 - Additional Instances 715
 - Commit Count 715
 - Commit Interval 715
 - Coordinated Transaction 715
 - configuration for coordination transactions 93
 - databases 96
 - nodes 93
 - WebSphere MQ and RRS 106
 - connections 8
 - adding with the mouse 141
 - adding with the Terminal Selection dialog 141
 - removing 142
 - conversion exception trace output 711
 - coordination 51
 - database connections 734
 - database support 734
 - copying 127
 - correcting save errors 134
 - creating 126
 - creating ESQL code 158
 - customizing nodes with ESQL 168
 - data conversion 107
 - data integrity 702
 - data types 735
 - BLOB message 739
 - DestinationData subtree 736
 - headers 735
 - MRM message 738
 - Properties subtree 736
 - database
 - connections 700
 - listing connections 701
 - database exception trace output 709
 - debugging 451
 - default version 479
- message flows (*continued*)
 - defining content 135
 - deleting 130
 - description properties 479
 - keywords 479
 - designing 59
 - destination lists
 - creating 76
 - using to route messages 71
 - developing
 - MIME 40
 - displaying version and keywords 131
 - errors 111
 - catching in TryCatch 120
 - connecting failure terminals 113
 - input node 114
 - MQInput node 115
 - Timeout Notification node 119
 - ESQL 146
 - exception list structure 707
 - exceptions, catching in TryCatch 120
 - field names, IDoc parser 740
 - globally coordinated transaction 51
 - input nodes
 - defining characteristics 68
 - using more than one 68
 - keywords
 - description properties 479
 - guidance 480
 - logical message tree
 - viewing 81
 - lost messages, avoiding 109
 - managing 473
 - managing ESQL files 156
 - message content, testing 70
 - message parser element
 - definitions 735
 - message structure, testing 69
 - MIME
 - message details 37
 - tree details 39
 - moving 129
 - nodes 4
 - adding with the GUI 135
 - adding with the keyboard 136
 - aligning 144
 - arranging 144
 - configuring 138
 - configuring for coordinated transactions 93
 - connecting with the mouse 141
 - connecting with the Terminal Selection dialog 141
 - deciding which to use 60
 - decision making 69
 - removing 139
 - renaming 137
 - opening 127
 - order, imposing 70
 - Parse Timing property 706
 - parser exception trace output 713
 - parsers 26
 - BLOB 46
 - IDoc 34
 - JMS 34
 - MIME 35
- message flows (*continued*)
 - parsers (*continued*)
 - MQCFH 741
 - MQCIH 742
 - MQDLH 743
 - MQIHH 743
 - MQMD 744
 - MQMDE 745
 - MQRFH 745
 - MQRFH2 746
 - MQRMH 746
 - MQSAPH 747
 - MQWIH 747
 - MRM 28
 - SMQ_BMH 747
 - XML 29
 - porting considerations 716
 - preferences 479
 - projects 4
 - creating 123
 - deleting 124
 - managing 122
 - promoted properties 49
 - converging 371
 - promoting 365
 - removing 370
 - renaming 369
 - properties 49
 - redeploying 473
 - relationship with ESQL and mappings 6, 303
 - removing from an execution group 445
 - renaming 128
 - response time, optimizing 73
 - restrictions for code page GB18030 699
 - save errors, correcting 134
 - saving 132
 - saving as 134
 - shared queues 78
 - subflows 4
 - adding 136
 - configuring 138
 - keywords 72
 - removing 139
 - renaming 137
 - using 71
 - supported code sets 671
 - system considerations 75
 - terminals 8
 - timeout control
 - automatic messages 406
 - multiple messages 405
 - sending a message 404
 - sending messages at a specified time 405
 - transaction support 51
 - TryCatch
 - catching exceptions 120
 - user database
 - DBCS restrictions 702
 - quiescing 701
 - user exception trace output 713
 - user exits 380
 - deploying 383
 - developing 383

- message flows (*continued*)
 - user-defined nodes 671
 - user-defined properties 50
 - validating messages 79
 - validation properties 703
 - version and keywords 7
 - WebSphere MQ connections 699
 - z/OS data sources
 - Compute node 778
 - Database node 778
- Message Mapping editor 748
 - Edit pane 754
 - Source pane 749
 - Spreadsheet pane 755
 - Target pane 752
- message sets
 - broker archive (bar) file, adding to 432
- messages
 - debugging 470
 - parsing on demand 706
 - partial parsing 706
 - self-defining and predefined 25
 - test message, getting 461
 - test message, putting 459
 - validating
 - Fix property 703
 - in message flows 79
 - Include All Value Constraints property 703
 - Validate property 703
- migration
 - mappings 759
 - restrictions 760
- MIME
 - message details 37
 - message flows 40
 - tree details 39
 - Web service 40
 - HTTP transport 41
 - WebSphere MQ transport 42
- MQeInput node 573
- MQeOutput node 580
- MQGet node 584
- MQInput node 593
- MQJMSTransform node 602
- MQMD
 - accessing with ESQL 189
- MQOptimizedFlow node 603
- MQOutput node 605
 - using in debugging 461
- MQReply node 612
- MQRFH2
 - accessing with ESQL 190

- N**
 - nodes, stepping over using the flow debugger 467
 - NULL handling
 - XML parser 33
 - numeric order in data conversion 92

- O**
 - object keyword 425
 - object version 425
 - order
 - imposing within a message flow 70
 - Output node 615

- P**
 - parsers 26
 - BLOB 46
 - choosing 47
 - IDoc 34
 - JMS 34
 - MIME 35
 - MQCFH 741
 - MQCIH 742
 - MQDLH 743
 - MQIIH 743
 - MQMD 744
 - MQMDE 745
 - MQRFH 745
 - MQRFH2 746
 - MQRMH 746
 - MQSAPH 747
 - MQWIH 747
 - MRM 28
 - null handling 46
 - partial parsing 48
 - SMQ_BMH 747
 - XML 29
 - NULL handling 33
 - Passthrough node 617
 - performance
 - message flow response time 73
 - populate 307
 - predefined messages 25
 - preferences
 - flow debugger 456
 - projects
 - message flows 4
 - promoted properties, message flow 49
 - Properties folder 15
 - Properties tree
 - accessing with ESQL 190
 - properties, message flow 49
 - Publication node 619

- Q**
 - quiescing databases 701

- R**
 - Real-timeInput node 621
 - Real-timeOptimizedFlow node 623
 - renaming deployed objects 445
 - repeating elements, configuring mappings 315
 - ResetContentDescriptor node 626
 - RouteToLabel node 630

- S**
 - SCADAInput node 632
 - SCADAOutput node 639
 - schemas
 - broker 52
 - self-defining messages 25
 - server project, creating 430
 - setting accounting origin 376
 - snapshot data 55
 - source code
 - stepping into 469
 - stepping out of 469
 - stepping over 468
 - statistics and accounting data 54
 - accounting origin 56
 - collecting 374
 - collection options 55
 - output formats 57
 - parameters, modifying 379
 - parameters, viewing 378
 - resetting archive data 380
 - setting accounting origin 376
 - starting 374
 - stopping 377
 - subflows
 - keywords 72
 - stepping into 467
 - stepping out of 467
 - symbols for flow debugger 994

- T**
 - terminals, message flows 8
 - test messages
 - getting 461
 - putting 459
 - Throw node 642
 - timeout control
 - automatic messages 406
 - multiple messages 405
 - sending a message 404
 - sending messages at a specified time 405
 - Timeout Control node 644
 - Timeout Notification node 647
 - error handling 119
 - timeout request message 402
 - topics
 - deployment 427
 - topics hierarchy, deploying 439
 - topology
 - deploying 438
 - deployment 426
 - Trace node 652
 - trademarks 1001
 - TryCatch node 656

- U**
 - user databases
 - accessing 84, 86, 88
 - configuring coordinated message flows 96
 - user-defined nodes 671
 - user-defined properties, message flow 50

V

- Validate node 658
- version
 - default value 479
 - displaying 131
- version and keywords
 - message flows 7

W

- Warehouse node 661
- WebSphere MQ
 - connections 699

X

- XML self-defining message
 - AttributeDef 772
 - AttributeList 772
 - DocTypeComment 773
 - DocTypeDecl 769
 - DocTypePI 774
 - DocTypeWhiteSpace 774
 - document type declaration 769
 - DTD 769
 - example 774
 - ElementDef 772
 - example message 762
 - external DTD 769
 - inline DTD 769
 - message body 765
 - AsIsElementContent 765
 - Attribute 766
 - BitStream 766
 - CDataSection 766
 - Comment 767
 - Content 767
 - Element 767
 - EntityReferenceEnd 768
 - EntityReferenceStart 768
 - example 768
 - ProcessingInstruction 768
 - NotationDecl 770
 - WhiteSpace 774
 - XML declaration 763
 - example 764
 - XML entities 770
- XMLTransformation node 665
- XPath
 - using 295
- XSL stylesheet
 - keywords 670

Z

- z/OS
 - data sources
 - Compute node 778
 - Database node 778



Printed in USA